PCJ. Parallel Computing in Java

Marek Nowicki, Piotr Bała
ICM UW, WMiI UMK

November 22, 2013

Contents

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 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. Than the internode communication has been added allowing users to run multiple PCJ threads withon single Java Virtual Machine. Current version has been developed in 2013 and includes many bug fixes and improvements compare to the initial version. Especially

the users interface has been stabilized.

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 tasks and are stored in the local memory.

Shared variables There is dedicated class called *Storage* which represents shared memory. Each task can access other tasks variables that are stored in a shared memory. Shareable variable has to have a special annotation `@Shared`.

There is distinction between nodes and tasks (PCJ threads). One instance of JVM is understood as 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. In contrast to our previous version of the PCJ library, 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 node. PCJ library takes care on this process and allows 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 resonable

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.

Since 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 pick up proper data exchange mechanism. Moreover, nodes are organized in the graph which allows to optimize global communication.

Chapter 3

PCJ basics

In order to use PCJ library you have to dowload `pcj.jar` file from the PCJ web site: **pcj.icm.edu.pl**. The `pcj.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 extends `Storage class` and implements `StartPoint` interface. The `Storage class` can be used to define shared variables. `StartPoint` interface provides necessary functionality to start required threads, enumerate them and performs initial synchronization of tasks.

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

```
10 import pl.umk.mat.pcj.PCJ;
11 import pl.umk.mat.pcj.StartPoint;
12 import pl.umk.mat.pcj.Storage;
13
14 public class PcjHelloWorld extends Storage implements StartPoint {
15
16     @Override
17     public void main() {
18         System.out.println("Hello!");
19     }
20
21     public static void main(String[] args) {
22         String[] nodes = new String[]{"localhost", "localhost"};
23         PCJ.deploy(PcjHelloWorld.class, PcjHelloWorld.class, nodes
               );
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.jar PcjHelloWorld.java
java  -cp .:pcj.jar PcjHelloWorld
```

The expected output is presented below:

```
PCJ version 2.0.0.161 built on Sun, 6 Oct 2013 at 14:50:44 CEST.
Starting PcjHelloWorld with 2 thread(s)...
Hello!
Hello!
```

The above scenario allows to run 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

PCJ library offers two useful methods:

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

  which returns number of tasks running and
- ```
  public static int PCJ.myId()
  ```

  which returns id of the task.

Task 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 tasks. While this line is reached, the execution is stopped until all tasks reach the synchronization line.

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

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

The user can provide argument to `barrier()` which is integer id of the task to synchronize.

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

In this case two tasks are synchronized: one with the given id and one which starts s`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 possibility to mark some variables `Shared` using Java annotation mechanism. The Shared variables are accessible across tasks, eg. one task can get access to the shared variable instance stored in another task.

```
40 @Shared
41 double a;
42
43 @Shared
44 double[] b;
```

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

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

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

```
double c;
if (PCJ.myId()==0) c =(double) PCJ.get(3, "a");
```

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

```
if (PCJ.myId()==0) PCJ.put(3, "a", 5.0);
```

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

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

3.5.1 get()

The get() method from PCJ library returns value of type `Object` 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 local variable is updated.

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

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

```
60 FutureObject aL;
61
62 if (PCJ.myId() == 0) {
63     aL = PCJ.getFutureObject(5, "a");
64     double a = (double) aL.get();
65 }
```

The remote value is transferred to the variable `aL` in asynchronous way. When data is available it is stored in the local variable `a`. This command is executed after local variable `aL` is updated.

3.5.2 put()

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.

```
70
71  @Shared double a;
72
73          if (PCJ.myId() == 0) {
74              PCJ.monitor("a");
75          }
76          if (PCJ.myId() == 2) {
77              PCJ.put(0, "a", 10.0);
78          }
79          if (PCJ.myId() == 0) {
80              PCJ.waitFor("a");
81          }
```

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 `b`.

3.5.3 broadcast()

In order to access variables at all tasks, PCJ provides 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.

```
80     @Shared double a
81     PCJ.monitor("a");
82
83     if (PCJ.myId() == 0) {
84         PCJ.broadcast("a", 2.14) ;
85     }
86
87     PCJ.waitFor("a");
88     System.out.println("a="+a);
```

In order to synchronize variables we set up monitor on the variable `a`. Than 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()`, `get()` 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 whole array as presented below.

```
90      @Shared  int[] array;
91
92      array = new int[20];
93      int[] c = new int[30];
94
95      PCJ.barrier();
96
97          if (PCJ.myId() == 0) {
98              c =  PCJ.get(3, "array");
99          }
100
101      PCJ.barrier();
102      System.out.println(array[4]);
```

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

```
90      @Shared int[] array;
91
92      array = new int[20];
93      int b = 0;
94
95      PCJ.barrier();
96      if (PCJ.myId() == 0) {
97          b = PCJ.get(3, "array", 6);
98      }
99
100     PCJ.barrier();
101     System.out.println(b);
```

3.6.2 put()

Simillar functionality van be achieved with `put()` method.

```
90      @Shared  int[] array;
91
92          array = new double[4];
93          if (PCJ.myId()==3){
94              array[3] = 30.0;
95           }
96
97          PCJ.monitor("array");
98          PCJ.barrier();
99
```

```
100         if (PCJ.myId() == 3) PCJ.put(3, "array", array);
101         if (PCJ.myId() == 0) PCJ.waitFor("array");
102 
103         System.out.println(PCJ.myId() + " ac " + 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 array in the broadcast is similar to the use of the simple variable.

```
90     @Shared double[] array
91 
92     double[] a = new double[] {0.577, 1.618, 2.718, 3.141}
93     PCJ.monitor("array");
94     PCJ.barrier();
95 
96     if (PCJ.myId() == 0) {
97         PCJ.broadcast("array", a );
98         }
99     PCJ.waitFor("array");
100    System.out.println(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 number of thread. However once PCJ application is run on multiple VM's, the detailed behavior depends on the mechanism used to launch application. IN many cases user will output from the local virtual machine.

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

```
    if (PCJ.myId()==0) System.out.println("Hello!");
```

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

The output using files could be performed in similar way. **This issue is discussed later.**

3.7.1 PCJ.log()

PCJ offers method to output information from each tasks. This method sends String argument to the thread 0 and performs output to the standard ooutput.

```
    PCJ.log("Hello!");
```

Example output while running with 3 threads looks like:

```
0 > Hello!
2 > Hello!
1 > Hello!
```

Please note that output is not serialized and output from different tasks is not ordered.

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 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.

```
120     @Shared int a;
121
122     Scanner stdin = new Scanner(System.in);
123
124     PCJ.monitor("a");
125     if (PCJ.myId()==0) {
126         a = stdin.nextInt();
127         PCJ.broadcast("a",a);
128     }
129     PCJ.waitFor("a");
130
131     System.out.println("a = "+a);
```

The input is performed by task0, therefore all other tasks 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 with 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 thread creates its own file handler and controls reads/writes from the file.

```
120     String b;
121     Scanner sc = null;
122     try {
123         sc = new Scanner(new File("input.txt"));
124     } catch (FileNotFoundException ex) { }
125     b = sc.next();
126     PCJ.log(" " + 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 idependently line by line. If one of threads reads more lines, threads can point to the different lines. In result read performed by all threads can retursn different values.

```
120     b = sc.next();
121     if (PCJ.myId() == 0) {
122         b = sc.next();
123     }
124     b = sc.next();
125     PCJ.log(" " + b);
```

Output is as following:

```
0 >   line3
1 >   line2
2 >   line2
```

3.10 Parallel reads from multiple file

The reading from 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 haevy access to the shared filesystem and can result in the performance decrease.

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

```
120     Scanner sc = null;
121      String f = "input"+PCJ.myId()+".txt";
122     try {
123         sc = new Scanner(new File(f));
124     } catch (FileNotFoundException ex) { }
125     b = sc.next();
126     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 performnce 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 the simillar way as input. User can either write data to the single file located on the shred 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 decreass performance, especially if it is located on the shared filesystem.

3.12 Java and PCJ library

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

```
301         double s = 0;
302         for (long i=0; i<n; i++){
303             s = s + Math.radom();
304         }
```

Listing 1: Random number generation with Math..random()

The above code will not scale while running multiple PCJ threads withiin single virtual machine, even if running on the multiprocesor/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 ising its own instance of the `Random` object which ensures parallel execution of all operations.

```
311 import java.util.Random;
312         double s = 0;
313         Random r = new Random();
314 
315         for (long i=0; i<n; i++){
316             s = s + r.nextDouble();
317         }
```

Listing 2: Random number generation with Random()

Plase remember that in this case instead of the single stream of the pseudorandom numbers, we are using mutiple strams of pseudorandom number which nor necessary has the same statistical properties[1].

Practical usage of this code you can find in the chapter 5.1 describing approximation of π with the Monte Carlo method.

[1]The generation of the pseudo random number in the parallel applications is well known problem which received significant number of publications. We will discuss this issue in the

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 abut the nodes parallel application will be running on is not available in advance or even during job submission but is determined when 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 miltiprocessor 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 other command to start parallel application.

nodes.txt - file containing list of 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.depoy() 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(MyStart.class, MyStart.class, "nodes.txt");
```

4.1 Linux Cluster

The user has to compile PCJ application with java (Java 7 and above required). Than the mpiexec command is used to run application. The user has to prepare files nodes.unique and nodes.txt as described above. The `mpiexec` 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 definition of the parameters passed to the queueining 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.

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.

Chapter 5

Simple examples

5.1 Hello world

Calculations start from a special *start point* class. That class contains main method (`public void main()`).

```
120 import pl.umk.mat.pcj.PCJ;
121 import pl.umk.mat.pcj.StartPoint;
122 import pl.umk.mat.pcj.Storage;
123
124 public class PcjHelloWorld extends Storage implements StartPoint {
125
126     @Override
127     public void main() {
128           System.out.println("Hello from " + PCJ.myId() + " of " +
                PCJ.threadCount());
129     }
130
131     public static void main(String[] args) {
132         String[] nodes = new String[]{"localhost", "localhost"};
133         PCJ.deploy(PcjHelloWorld2.class, PcjHelloWorld2.class,
              nodes);
134     }
135 }
```

: PcjHelloWorld2.java

The compilation and execution requires `pcj.jar` in the path:

```
javac -cp .:pcj.jar PcjHelloWorld.java
java  -cp .:pcj.jar PcjHelloWorld
```

The expected output is presented below:

```
PCJ version 2.0.0.164 built on Thu, 7 Nov 2013 at 14:16:04 CET.
   Starting PcjHelloWorld with 2 thread(s)...
Hello from 0 of 2
Hello from 1 of 2
```

Please note that especuially for large number of tasks, their numbers can appear in teh random order.

5.2 Sequential execution

In order to ensure sequential execution of code, ie. outputr from taks in given order, the `PcjHelloWorld,java` example should be modified. We introduce loop over the thread id, and the thred which number os equal to the loop variable is executing operations.

```
120     PCJ.barrier();
121     for (int p = 0; p < PCJ.threadCount(); p++) {
122         if (PCJ.myId() == p) {
123             System.out.println("Hello from " + PCJ.myId() + " of "
                    + PCJ.threadCount());
124         }
125         PCJ.barrier();
126     }
```

: PcjHelloWorldSequecial.java

5.3 Reduction

Reduction operation is widely used to gather values of some variable stored on different threads. In the presented example the values are communicated to the thread with the id 0. Than reduction operation such as summation is performed.

The local array `aL` is created at thread 0. Than value of the variable a stored at the thread p is communicated to the thread 0 and stored in the `aL[p]`. Finally, the reduction operation is performed on the values stored locally in the array `aL`. The `aL[p].get()` operation performed on the `FutureObject` guarantees that data is available on the thraed 0.

```
200     @Shared double a
201
202     FutureObject aL[] = new FutureObject[PCJ.threadCount()];
203     double a0 = 0.0;
204     if (PCJ.myId() == 0) {
205         for (int p = 0; p < PCJ.threadCount(); p++) {
206             aL[p] = PCJ.getFutureObject(p, "a");
207         }
208         for (int p = 0; p < PCJ.threadCount(); p++) {
209             a0 = a0 + (double) aL[p].get();
210            }
211        }
```

: PcjReduction.java

The presented algorithm of the reduction is based on the asynchronous communication since `PCJ.getFutureObject()` method is executed independently by threads. Summation is then performed as data arrives at the thread 0.

Chapter 6

Parallel applications

In this chapter we present some parrallel applications implemented with the PCJ.

6.1 Approximation of π using MonteCarlo

The program picks points at random inside the square. It then checks to see if the point is inside the circle (it knows it's inside the circle if $x^2+y^2 < R^2$, where x and y are the coordinates of the point and R is the radius of the circle). The program keeps track of how many points it's picked (`nAll`) and how many of those points fell inside the circle (`circleCount`).

In the parallel version, the work is divided among threads, i.e. each traed is performing $nAll/PCJ.threadsCount()$ attempts. Each thread counts points inside circle.

Finally, the number of caounted points has to be gathered by the thread 0. This is permormed as the simple reduction.

```
301 import java.util.Random;
302 import pl.umk.mat.pcj.FutureObject;
303 import pl.umk.mat.pcj.PCJ;
304 import pl.umk.mat.pcj.Shared;
305 import pl.umk.mat.pcj.StartPoint;
306 import pl.umk.mat.pcj.Storage;
307
308 public class PcjExamplePiMC extends Storage implements StartPoint
    {
309
310     @Shared long circleCount;
311
312     @Override
313     public void main() {
314
315         long nAll = 100000000;
316         long n = nAll/PCJ.threadCount();
317         Random r = new Random();
318
319         circleCount = 0;
320         double time = System.nanoTime();
321 // Calculate
322         for (long i=0; i<n; i++){
```

```
323             double x = 2.0 * r.nextDouble() - 1.0;
324             double y = 2.0 * r.nextDouble() - 1.0;
325             if ( (x*x + y*y) < 1.0 ) circleCount++;
326         }
327         PCJ.barrier();
328 // Gather results
329         long c = 0;
330         FutureObject cL[] = new FutureObject[PCJ.threadCount()];
331 
332         if (PCJ.myId() == 0) {
333             for (int p = 0; p < PCJ.threadCount(); p++) {
334                 cL[p] = PCJ.getFutureObject(p, "circleCount");
335             }
336             for (int p = 0; p < PCJ.threadCount(); p++) {
337                 c = c + (long) cL[p].get();
338             }
339         }
340 // Calculate pi
341         double pi = 4.0 * (double) c / (double) nAll;
342         time = System.nanoTime() - time;
343 // Print results
344         if (PCJ.myId() == 0 ) System.out.println(pi + " " + time *
                1.0E-9);
345       }
346 
347     public static void main(String[] args) {
348         String[] nodes = new String[]{"localhost","localhost","
              localhost","localhost"};
349         PCJ.deploy(PcjExamplePiMC.class, PcjExamplePiMC.class,
              nodes);
350     }
351 }
```

Listing 3: Approximation of π using Monte Carlo.

6.2 Approximation of π using integral

The listing presents whole source code of application that approximates π value. The value is calculated using rectangles method that approximates following integral:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$$

In our code, the interval is divided into 1000 equal subintervals and we take top middle point of each subinterval to calculate area of the rectangle.

The calculations will start by executing the main method from MyStartPoint class with MyStorage class as storage. Four tasks will be involved in calculations: one on local machine, two on *node1* and last one on *node2*. The listing contains comments that should clarify what program is doing. The user can easily change number of tasks by providing more host names to the `deploy` method. The PCJ will launch calculations on specified nodes.

```
301 iimport java.util.Locale;
```

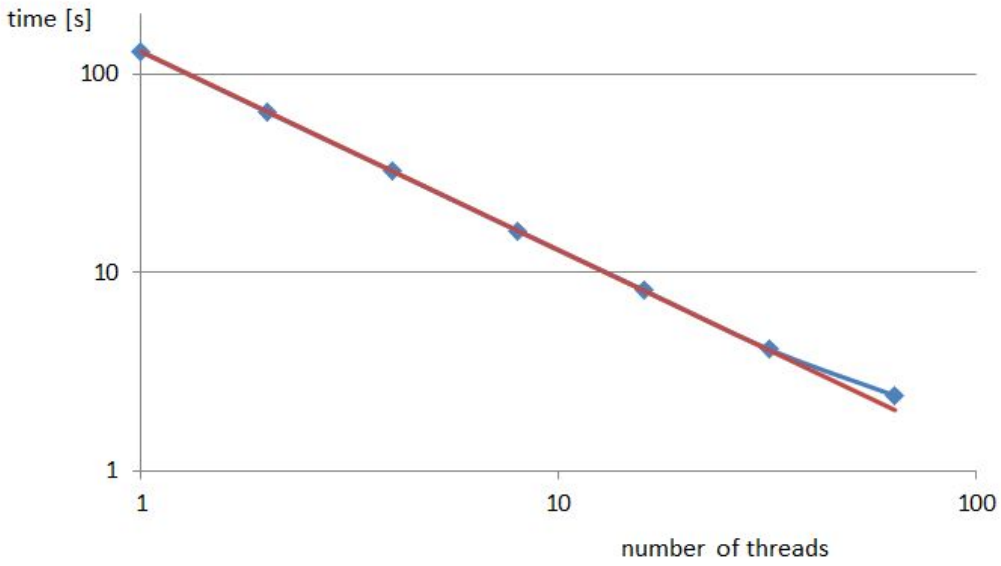

Figure 6.1: The performance of the code to approximate π using Monte Carlo method. The code has been executed on the PC cluster halo2 at ICM. The red line shows ideal scalling.

```
302 import pl.umk.mat.pcj.PCJ;
303 import pl.umk.mat.pcj.Shared;
304 import pl.umk.mat.pcj.StartPoint;
305 import pl.umk.mat.pcj.Storage;
306 import pl.umk.mat.pcj.FutureObject;
307
308 public class PcjExamplePiI extends Storage implements StartPoint {
309
310     private double f(final double x) {
311         return (4.0 / (1.0 + x * x));
312     }
313
314     @Shared
315     double sum;
316
317     @Override
318     public void main() {
319         PCJ.barrier();
320         double time = System.nanoTime();
321
322             long nAll = 1000;
323             double w = 1.0 / (double) nAll;
324             sum = 0.0;
325
326             for (int i=PCJ.myId(); i<nAll; i+=PCJ.threadCount()) {
327                 sum = sum + f(((double) i + 0.5) * w);
328             }
329             sum = sum * w;
330
331         PCJ.barrier();
```

```
332
333         FutureObject cL[] = new FutureObject[PCJ.threadCount()];
334
335         double pi = sum;
336         if (PCJ.myId() == 0) {
337             for (int p = 1; p < PCJ.threadCount(); p++) {
338                 cL[p] = PCJ.getFutureObject(p, "sum");
339             }
340             for (int p = 1; p < PCJ.threadCount(); p++) {
341                 pi = pi + (double) cL[p].get();
342             }
343         }
344
345         PCJ.barrier();
346
347         time = System.nanoTime() - time;
348
349          if (PCJ.myId() == 0) {
350             System.out.format(" %d  %f7 time %f5 \n", pi, time *
                   1.0E-9, time);
351         }
352     }
353
354     public static void main(String[] args) {
355         String[] nodes = new String[]{"localhost", "localhost"};
356         PCJ.deploy(PcjExamplePiI.class, PcjExamplePiI.class, nodes
               );
357     }
358
359 }
```

: PcjExamplePiI.java caption

Acknowledgments

This work has been performed using the PL-Grid infrastructure.

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/