PCJ-Parallel Computing in Java

Mansur Alizada
Institute of Computer Science
University of Tartu
Email: alizada@ut.ee

***Abstract*—PCJ is a library 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.**

I. Introduction to PCJ

With the widespread adoption of multi-core and multi-processor system parallel programming is not a trivial task. The automatic tools cannot be used, since the parallelization of the problem is performed on the algorithmic level. Both the domain expertise and computer science knowledge is necessary to develop parallel algorithms. Their implementation is very complex, mainly due to the parallel tools, libraries and programming models. The message passing model is an example of the more difficult ones. While the knowledge of shared memory model is easy to acquire, it requires the ability to write well-scaled codes. Others are appropriate only for a certain class of problems, an example of which is Map-Reduce.

New languages such as Java, Python and Scala are nowadays substituting traditional languages such as FORTRAN and C/C++. Most implementations have their basis in those traditional languages. The efficient implementation of parallel algorithms can be possible also in PGAS programming model.

II. PCJ Library

PCJ is a library of the Java language [2, 3, 4, 5] whose aim is to help with performing parallel and distributed computations. It possesses the ability to work on the multicore systems with the typical interconnect (such as ethernet or infiniband). Thus, it provides users with the uniform view across nodes. The library is OpenSource (using BDS license). Its source code can be accessed at GitHub. It is capable of working with multi-core systems with typical interconnection provided by ethernet or infiniband users with a unified view of nodes. library is OpenSource (BSD license), and its source code available from GitHub. Being inspired by languages like Co-Array Fortran [6], Unified Parallel C [7] and Titanium [11], PCJ implements partitioned global address space model.

One of the greatest benefits provided by PCJ is that there is no need to use other libraries outside of the standard Java distribution, since it does not modify language syntax. This is a unique characteristics compared to previously mentioned languages.

As can be seen in Figure 1, there is a separate local memory for each PCJ thread (task), which produces its own set of instructions. Variables of other tasks can be accessed if those include a special annotation @Shared. Additionally, the methods for basic operations (synchronization of tasks, get and put values in an asynchronous one-sided way) are provided.

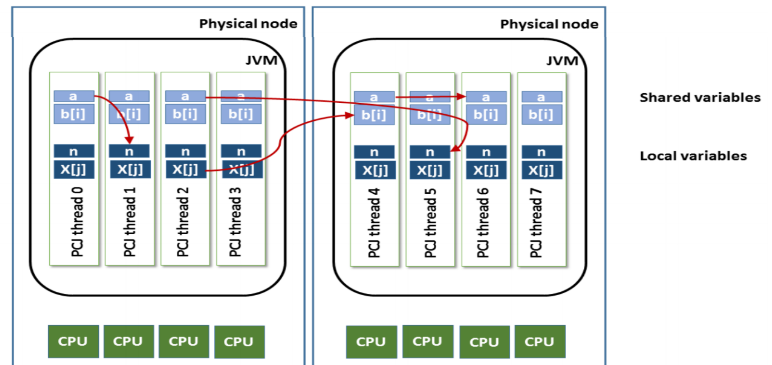

Figure 1: The scheme of PCJ threads

Applications which use PCJ library are run using Java Virtual Machine (JVM). In the multinode environment, PCJ library allows one or more JVM to be started on each node. The communication manners between different threads vary, since PCJ application is not running within single JVM. On the

contrary, communicating threads running withing the same JVM, it is necessary to use concurrency mechanisms to synchronize and exchange information.

III. PCJ DETAILS

The basic primitives of PGAS programming paradigm offered by the PCJ library are as follows and may be executed over all the threads of execution or only a subset forming a group:

get(int threadId, String name) - allows to read a shared variable (tagged by name) published by another thread identified with *threadId*); both synchronous and asynchronous read with FutureObject is supported;

put(int threadId, String name, T newValue) - dual to get, put writes to a shared variable (tagged by name) owned by a thread identified with *threadId*; the operation is non-blocking and may return before target variable is updated;

barrier() - blocks the threads until all pass the synchronization point in the program; a two-point version of barrier that synchronizes only the selected two threads is also supported

broadcast (String name, T newValue) - broadcasts the *newValue* and writes it to each threads shared variable tagged by name;

waitfor(String name) - due to the asynchronicity of communication primitives a measure that allows one thread to block until another changes one of its shared variables (tagged with a name) was introduced.

Due to above presented methods, complex parallel algorithms can be implemented. However, constructs for automatic data distribution are not provided in the PCJ library. Therefore, this task has to be carried out by the programmer.

IV. NODE NUMBERING

A node used for setting unique identifiers to the tasks, sending messages to other tasks to start calculations, creating groups an synchronizing all tasks in calculations is called *Manager*. *The Manager* node is the *Master* of a group in all tasks. It has a group identifier which equals 0. Identifiers in each node are unique for whole calculations. That node is called *physical id*. Before starting a calculation, all nodes are interconnected. In this process, nodes exchange their *physical node ids*.

Should user wants to start using PCJ for parallel execution, he or she has to execute static method *PCJ.start()* providing information about requested StartPoint and Storage classes and list of nodes. The list is used in order to provide numbering of PCJ nodes and PCJ threads. The list is processed by every PCJ node to localize items containing its hostname data. Subsequently, PCJ threads are numbered using items numbers.

A special node that is used for coordinating other nodes in a startup is called *node0* and is listed in the first place of the list. After the list is processed, every node connects to *node0* and tells the items numbers from the list, that contains its hostname. When the information about every node from the list is obtained by *node0*, it starts numbering the nodes with numbers form 0, increasing the number by one on each distinguished node. The number is called *physicalId*. *Node0* gives response to all the remaining nodes with their *physicalId*.

After receiving their *physicalId*, nodes have to exchange information and connect to each other. In order to achieve this step, *node0* is broadcasting information about each node using a balanced tree structure, where each node contains two children at maximum. *Node0* represents the root and it is the only one vertex of the tree at the beginning. The following information is broadcast about each new node in that tree: *physicallId*, parent *physicallId*d, *threadIds* and *hostname*.

Everytime this information is received, it is sent down the tree and saved. When a node is the parent of the new node, it adds it as own children. Subsequently, the connection to a new node is made and the node sends information about itself (*physicalId* and *threadIds*). The initialization step is completed at the end, when *node0* receives the information about all nodes with the physical id except for physical id of the new node.

Node0 sends a message to start user application once all nodes send information about completion of the initialization step. Each node starts adequate number of PCJ threads using the *StartPoint* class that was provided.

V. Communication

There is a distinction between inter- and intranode communication in the PCJ library, which is able to pick up proper data exchange mechanism, depending on the task ids involved in the communication.

Java New IO classes (java.nio.*) are used to perform the network communication between individual nodes. It is possible that data exchange between PCJ threads is asynchronous. Sending a value to another task storage is using the *put* method (see Figure 2). Due to the asynchronic data transfer, there is also the *waitFor* statement executed by the PCJ thread receiving data. For obtaining value from other task storage, the *get* method is used. There is also the *getFutureObject* method, which - unlike the previous two methods - works in fully nonblocking manner.

```
@Shared
double a;
double c=5.0;
if(PCJ.myId()==i ){
    PCj.put(j,"a",c);

}
  if(PCJ.myId()==j){
    PCJ.waitFor( variable: "a");
  }
```

Figure 2: PCJ put method

VI. Broadcast

As described above, *Node0* uses a tree structure to broadcast messages to all nodes. This message includes the value serialized by source PCJ. After the message is received, it is sent down the tree, deresialized and stored into specified variable for all PCJ thread storages. An instance of the broadcast is provided in Figure 3.

```
10 @Shared
11 double a;
12
13 double c = 10.0;
14 if (PCJ.myId() == 0 ) {
15       PCJ.broadcast("a", c);
16 }
```

Figure 3: PCJ broadcast

VII. Synchronization

During synchronization, a proper message is sent to the group master by one task. When this is carried out by all tasks, the group master uses the binary tree structure to send an adequate message to all tasks. An example of the PCJ synchronization of the execution performed by all PCJ threads can be: *PCJ.barrier();*.

Furthermore, it is possible that two threads can synchronize their execution in a following way: a message is sent from one PCJ thread to another. The PCJ thread then waits for the response (the same message). When the message comes before the thread starts to wait, the execution is not suspended at all.

VIII. Fault tolerance

Additionally, in the PCJ library basic resilience mechanism can be found. Due to this fact, the programmer is provided with the basic functionality and therefore, is able to detect node failure. The Java exception mechanism is used for performing this task. Programmer is then presented with all detected execution problems associated with all intranode communication. Subsequently, the corresponding actions might be taken to continue program execution. It is the programmer who decides and implements the algorithm to recover from the failure.

The fault-tolerance implementation is based on the assumption that node 0 never dies. The probability of its failure is significantly smaller than the probability of one of other nodes and can be

neglected here. When a nod fails, node 0 is waiting for the hearthbeat message from that particular node. If such message is not received, it is assumed that the node died.

IX. PCJ Examples

PCJ has been used to parallelize the problem of a large graph traversing. In particular, it is sed to implemented Graph500 benchmark and evaluated its performance. Another example is parallelization of the differential evolution on example mathematical function as well as was to fine-tune the parameters of nematodes C. Elegans connectome model.

X. Sorting with PCJ

I used Bubble Sort algorithm for parallel sorting with PCJ.Using paralleling techniques with PCJ,allows us exchange values,saving them on the processor,compare them and etc.Using Parallel generalization of the operation,i sorted the block on each processor at the beginning of sorting.Then, exchanged the blocks between the processors.Then combined the blocks and on each processor into a sorted block with the help of merge operation.In the sorting stage,i separated phases to odd and even.Checking conditions,putting default phases for default PCJ numbers.In the end of the sorting,used swap operation for the numbers.Before beginning to sort,i defined 2 methods,for combining 2 part of the numbers due to bubble sort,comparing and deciding which part will go to the left and right.In my main method,it computes time in ns and returns maximum and minumum time for the random numbers.In the sorting everytime,after putting values inside of PCJ,PCJ.waitFor() methods defining the numbers accessed to PCJ numbers.

```
if (odd_rank < PCJ.threadCount() && odd_rank >= 0) {
    PCJ.put(odd_rank, "PCJ_numbers", random_numbers);
    PCJ.waitFor("PCJ_numbers");}
```

Figure 4: PCJ Put Method

Using PCJ's PCJ.put() method,defined correct values for the further process,which will define to merge numbers within sorted way.As the result we can see the figure below.

PCJ with 4 threads	
Data Size	**Time(in microsecond)**
10000000	37630.874
20000000	82591.355
30000000	132665.795
40000000	180648.812

Figure 5: PCJ Performance Table with data.

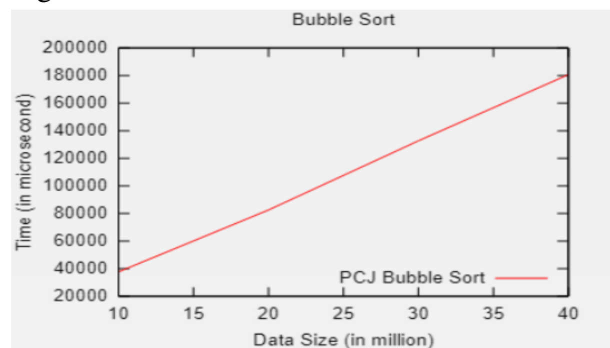

Figure 6: PCJ Performance Graph with data.

As we can see,from the table and the graph,although data size increases 2 times and time is increasing as well. With the help of shared variables,we use these variables inside of the class in every thread. With 4 threads result is like in the picture.

```
Starting javaapplication4.algo.JavaApplication4 with 4 thread(s)...
Max time is:  11241733 ns Min time is : 500295 ns
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 7: Result

Although,PCJ beats MPI in running time,however,PCJ needs a better fault tolerance.Self-stabilization of the PCJ is not well-built.Some phase of the running time,it can suddenly slow down,or shut down the threads and processes.Redundancy and replication might be useful in this stage for PCJ.

XI. Conclusion

The obtained results show good performance and good scalability of the benchmarks and applications implemented in Java with the PCJ library. The communication and synchronization cost is comparable

to other implementations such as MPI resulting in good performance and scalability.

XII. References

[1] D. Mallon, G. Taboada, C. Teijeiro, J.Tourino, B. Fraguela, A. Gomez, 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

[2] http://pcj.icm.edu.pl Accessed: 24.04.2018. Recent Advances in Information Science ISBN: 978-1-61804-365-8 71

[3] M. Nowicki, P. Baa. 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

[4] M. Nowicki, P. Baa. PCJ-new approach for parallel computations in java In: P. Manninen, P. Oster (Eds.) *Applied Parallel and Scientific Computing, (LNCS 7782)*, Springer, Heidelberg (2013) pp. 115-125

[5] M. Nowicki, . Gorski, P. Grabarczyk, P. Baa. PCJ - Java library for high performance computing in PGAS model In: W. W. Smari and V. Zeljkovic (Eds.) *2014 International Conference on High Performance Computing and Simulation (HPCS)*, IEEE 2014 pp. 202-209

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

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