Sindh Univ. Res. Jour. (Sci. Ser.) Vol. 48 (4) 913-918 (2016)

SINDH UNIVERSITY RESEARCH JOURNAL (SCIENCE SERIES

Comparative Analysis of Performance Achieved with Multi-Core Programming Models in Java

M. I. LALI, W. ASLAM*, F. ULLAH, M. S. NAWAZ**, S. RAZZAK, S. ZAHOOR

Department of Computer Science and IT, University of Sargodha. Pakistan.

Received 07th May 2016 and Revised 14rd October 2016

Abstract: In modern processor designs, physical limitations such as energy consumption and communication have led to the development of multi-core processors. Nowadays, different parallel programming models are used in order to fully utilize the capabilities of multi core systems. However, the overall influence of these models on performance and usability is still unclear. In this article, two parallel programming models (MPJ-Express and P2P-MPI) for Java are compared by implementing a sorting algorithm (bucket sort). Parallel programs of sequential bucket sort (in Java) are first developed in both models and comparative analysis of P2P-MPI and MPJ-Express is then carried out in terms of speedup and efficiency achieved in each model and effort required in conversion of sequential programs to parallel programs. Obtained results showed that MPJ-Express perform better than P2P-MPI for sorting algorithm.

Keywords: MPJ-Express, P2P-MPI, bucket sort, speedup, efficiency.

1. INTRODUCTION

Recent advancements and emerging trends in Information and communication technologies require computing devices with better computation and performance capabilities. Till 2005, fast processors were the main contributors towards a high-speed computer; however this is not the case anymore. Physical limitations in modern processor design such as energy consumption, heating, dispersion and communication delays have led to the development of multi-core processors. Multi-core processors are used to overcome the drawbacks faced earlier by putting into practice more than one processor for computation. Multi-core systems provide the opportunity to execute code blocks of the same program simultaneously. The need for simple and efficient parallel programming models for parallel computation is gaining more and more attention due to the ease of its accessibility (Nanz *et al.,* 2013).

For software developers, the introduction of multi-core systems means revolutionizing the way a program should be carved. Applications at the moment needs to be improved, upgraded and be able to run on multi-core processors that are evolved from new computational fields. The matter under discussion highlights the desire for sequential programs to be transformed for parallel execution on multi-core machines. Various models are available that can be used to parallelize the sequential programs. These models differ from each other in several ways such as syntax, scalability and tools due to exploiting threads or processes, shared memory or channels (network), portability, etc (Wahlen, 2010). Consequently, it is important to choose a model that performs well on hexa to octa-core processors as well as on new computing devices of many-core processors (Liest *et al.,* 2014).

For C/C++ language, different parallel programming models and libraries are now available which help programmers and developers to switch from sequential programming to parallel programming. These include Cilk Plus (Robinson, 2013), Threading building blocks (TBB) (Rienders, 2007), OpenMp (Chandra *et al.,* 2001), and Pthreads (Nicholas *et al.,* 1996). In literature, many ideas have been proposed that evaluate and compare different parallel programming models of C/C++. Wahlen (2010) compared four parallel programming models of C/C++. Comparison process carried out on account of speedup and efficiency by converting four sequential programs (Primes calculation, Adaptive quadrature, Post office simulation and Game of life) into parallel programs in each model. Tristram and Bradshaw (2010) used Mandelbort algorithm to compare three famous parallel programming models, TBB, OpenMP and Pthreads for C++. They showed that OpenMP requires the least coding efforts and Pthreads performed best. Jarp *et al.,* (2012) compared the performance of three models TBB, OpenMP and Cilk Plus on an algorithm for high energy physics data analysis. TBB gave the best results as compared to Cilk Plus and OpenMP in terms of performance with the minimum number of changes required in C++ application. Saule and Catalyurek (2012) compared the performance of irregular graph

++Corresponding author's email drikramullah@uos.edu.pk
*Department of Computer Science & IT, The Islamia University of Bahawalpur. Pakistan
**Department of Information Science, School of Mathematics, Peking University, Beijing, China

algorithms in TBB, OpenMP and Cilk Plus. Results showed that TBB, OpenMP and Cilk Plus can be used effectively to hide latencies in irregular applications for achieving almost perfect speedup.

Krieger *et al.,* (2012) discussed asynchronous parallelism that is represented with task graphs. They compared sparse matrix solver and a molecular dynamics benchmark in TBB. Podobas *et al.* (2010) offered a quantitative study for comparison of OpenMP, Cilk Plus and Wool. Results indicated that OpenMP works well for coarse grained application, whereas, Cilk Plus and Wool performed better for fine grained applications. Leist and Gilman (2014) investigated the performance of TBB, Cilk Plus and OpenMP for three algorithms: merge sort, matrix multiplication and dot product. In our previous work (Saleem *et al.,* 2014), we have developed parallel programs for quick sort and merge sort in Cilk Plus in order to evaluate the performance and estimated speedup achieved over the sequential quick and merge sort. Furthermore in (Nawaz, *et al.,* 2014), Cilk Plus, TBB, OpenMP and Pthreads are evaluated by implementing parallel quick sort program in each model. Results showed that Cilk Plus performed better for parallel quick sort program.

In software industry, Java programming language is now a leading platform for software development due to its ease of use, portability, high programming productivity and built-in support for multithreading and networking. Two popular parallel programming models for JAVA are MPJ-Express (Zafar *et al.,* 2015) and P2P-MPI (Gropp, *et al.,* 1996). These two models cover a wide range of programming abstractions for parallelism and are under active development supported by large corporations. In this article, MPJ-Express and P2P-MPI is compared by implementing bucket sort algorithm in each model. Comparison is based on the speedup and efficiency achieved in each model and to what extent efforts are required for conversion of sequential bucket sort program into parallel program in respective models.

The rest of the article is organized as follows: Section 2 gives an overview of MPJ-Express and P2P-MPI. In Section 3, bucket sort algorithm is discussed. Furthermore, sequential bucket sort program is converted into parallel program employing MPJ-Express and P2P-MPI. Experiments are performed in Section 4, where execution time of parallel bucket sort programs on different number of processors is calculated. Moreover, speedup and efficiency achieved by parallel programs in eachmodel is reported. Strengths and weaknesses of each model are also discussed in section 4. Results obtained leads to choose a suitable model for high performance according to usability and nature of the problem. Finally, conclusions are drawn in Section 5.

2. OVERVIEW ON MPJ-EXPRESS AND P2P-MPI

Software developers must exploit and generate parallelism in their programs as much as possible in order to get the full capabilities of multi-core systems. However, parallel programming has several implications as the development of parallel programmed software is hard and requires more expertise as compared to sequential programming (Hayes, 2007). Due to concurrency, programs are more vulnerable to errors, data races, deadlocks and on the other hand, performance enhancement is a big challenge as scheduling and communication overheads may lead to diverse effects. In response, various advanced programming models and libraries have been developed recently (Nanz *et al.,* 2013). These models are based on different programming abstractions. Two eminent parallel programming models for JAVA are described**.**

MPJ-Express: MPJ-Express is a message passing system that is used to set parallels the sequential Java codes on multicore processors. It is an open source MPI with Java messaging library with free implementation of the MPI Java 1.2 API. Practicality of various applications has been tested through it. There are two modes in MPJ Express software: multi-core mode and the cluster mode. In multi-core mode, parallel programs run on either multicore processors or shared memory. In cluster mode, programs are parallelized using MPJ Express. Parallel programs can be run on distributed memory platforms such as compute clusters and clouds.

Java developers use MPJ Express to implement their parallel Java application on compute clusters or network of personal computer. Computer clusters, a collection of loosely or tightly coupled computers used for large scale computational work, is a renowned parallel platform which is broadly used by the High Performance Computing (HPC) (Taboada, et al., 2013). MPJ Express is ultimately a middleware that supports communication among each processor of the network. MPJ express follows the Single Program Multiple Data (SPMD) programming model.

MPJ-Express monitors the execution of a program. For speed estimation of more than one processor, it monitors the single execution of the program by a predefined input data. In general, diverse data will cause diversity in the performance of the program. Results of MPJ-Express are shown on Eclipse (command prompt window) after program executes. It exhibits graphical representation of predicted speed and parallelism as well.

P2P-MPI: P2P-MPI provides a grid environment for execution of parallel programs. It offers middleware with the characteristics of self-configuration, data management, robustness of hosted processes, and abstract computing capability simultaneously. P2P-MPI offers a platform for implementation of parallel programs on a set of hosts that are connected to each other. According to the perspective of parallelization, P2P-MPI runs an MPJ (MPI for Java) library for processes communicate during program execution. P2P-MPI also provides an original process launcher system. It provides access to other's CPU and let share possessed one with others.

P2P-MPI can be seen as an abstraction of three layer stack. Communication library is at the top which exposes an MPJ API followed by Middleware that provides different services to communication library through a set of modules.

3. BUCKET SORT ALGORITHM

Bucket sort algorithm (Kessler and Keller, 2007), also known as bin sort, contains array elements distributed into a number of buckets. Buckets are categorized and each element is placed in its respective bucket. Each bucket is sorted individually by implementing the recursive bucket sort algorithm. It is a class of distribution sort similar to radix sort, in which elements are sorted from the most to least significant figure. It is an impression of pigeonhole that can be utilized for comparison and therefore can also be taken as a comparison sorting algorithm. The working of bucket sort is as follows:

1. Create an array of 'buckets', which are empty initially.
2. **Scatter**: In the original array, put each object of the array in its respective bucket.
3. Sort each non-empty bucket.
4. **Gather:** Buckets are visited in order and all elements in the bucket are placed back in the original array.

Pseudo-code for the implementation of bucket sort algorithm (Silva and Buyya, 1999):

BUCKET_SORT (A)

1. $n \leftarrow$ length[A]
2. For $i = 1 \ldots n$ do
3. insert A[j] into B[msbits(A[j], k)]
4. For $i = 0$ to n-1 do
5. nextsort(B[j]);
6. return the concatenation of B[0],B[1], ., B[n-1]

Array[A] is depicted as a problem to be sorted in this algorithm. Number of buckets is denoted by *n*. The function *msbits(x,k)* returns the *k* most major bits of *x* (*floor(x/2^(size(x)-k))*). Various functions can be used for decoding the array of elements in array to *n* buckets. The *nextSort* function is a sorting function. It can be used to translate the letters from *A* to *Z*, or numbers from 0 to 25, or return the first or last character for sorting series**.**

Sequential program for bucket sort in Java has been implemented, which can be found at: https://github.com/saqibdola/MulticoreModelsJava.
Bucket sort implementation is done by dividing the array of size "*n*" into "*k*" buckets. Each bucket consists of value range total array size "*n*" divided by number of buckets "*k*". Then buckets are sorted with sorting algorithm that works well on the predictable data. Following are the main steps:

1. Peek the unsorted array and determine its maximum value

```
public static void sort(int[] a,
intmaxVal)     {int     []     bucket=new
int[maxVal+1];
```

2. Length of bucket must be max *Value + 1* where index of an array begins with 0

```
for (inti=0; i<bucket.length; i++) {
bucket[i]=0; }
```

3. Divide the array elements into bucket

```
  for (inti=0; i<a.length; i++) {
bucket[a[i]]++; }
```

4. Go through the bucket from start and pick out the element which must be already sorted in bucket.

```
intoutPos=0;
 for (inti=0; i<bucket.length; i++) {
 for (int j=0; j<bucket[i]; j++) {
     a[outPos++]=i; } }
```

Parallel versions in MPJ-Express and P2P-MPI are implemented subsequently.

Parallel Bucket Sort in P2P-MPI and MPJ-Express

Parallel bucket sort is implemented in both models by sorting "*n*" numbers in range *[1, 2,…, n]* using "*p*" number of processors. These *n* numbers are sorted with *p* processors by dividing the elements into *n/p* buckets.

```
 for(i = 0; i<NUM_BUCKETS+TEST_ARRAY_
SIZE; i++) { bucket_size[i] = 0;
    bucket_size_totals[i] = 0;
    process_bucket_distrib_ptr1[i]=0;
process_bucket_distrib_ptr2[i] = 0;}
```

These buckets are visited by processors "*p*" sequentially. Determine where the desired number is and load it to the top of the bucket.

```
 for(i = 0; i< TEST_ARRAY_SIZE; i++)
{
```

Table 2: Execution time of sequential and parallel bucket sort programs

Dataset	Sequential	Parallel P2P-MPI			Parallel MPJ-Express		
		2 P	4 P	8 P	2 P	4 P	8 P
8 M	7.6s	1.7 s	1.32 s	0.96 s	1.5 s	0.98 s	0.78 s
30 M	24.4s	6 s	5.35 s	4.38 s	4 s	3.2 s	2.75 s
100 M	60.3 s	21 s	19 s	18.5 s	15 s	13 s	11.5 s

```
if((test_index_array[i]/NUM_KEYS)==
   my_rank){
bucket_size[NUM_BUCKETS+i] =
key_array[test_index_array[i]%NUM_KEY
S];}}
```

Drop all elements and sort them into the appropriate buckets.

```
   for(i = 0; i< NUM_KEYS; i++) {
     key = key_array[i];
     key_buff1[bucket_ptrs[key       >>
shift] ++]   = key;}
```

Now find out how many keys are sent to every other processor by each processor. And determine the received displacement for buckets.

```
 MPI.COMM_WORLD.Alltoall(send_count,
0, 1,
  MPI.INT, recv_count,0, 1, MPI.INT);
  recv_displ[0] = 0;
  for(i = 1; i<comm_size; i++)
  recv_displ[i] = recv_displ[i - 1] +
       recv_count[i - 1];
MPI.COMM_WORLD.Alltoallv(key_buff1,0,
     send_count,MPI.INT,key_buff2, 0,
  recv_count,recv_displ,MPI.INT);
```

The first and last bucket numbers on all processors are multiplied with the buckets central size in order to get the minimum and maximum value of any key on individual processor.

```
 min_key_val                          =
process_bucket_distrib_ptr1  [my_rank]
<< shift;
     max_key_val =

((process_bucket_distrib_ptr2[my_rank
]+1)
<< shift)-1;

for(i=0;        i<max_key_val        -
min_key_val+1; i++) key_buff1[i] = 0;
```

Get the rank of each key.

```
  key_buff1[0] += m;
  for(i         =         min_key_val;
i<max_key_val;i++) {key_buff1[i + 1 -
min_key_val]   +   =   key_buff1[i   -
min_key_val];}
```

For sorting, determine which processors have the keys of lesser value. And shift it to the backwards.

```
  m = 0;
  for(k=0; k <my_rank; k++)
  for(i=
process_bucket_distrib_ptr1[k];
i<=process_bucket_distrib_ptr2[k];i++
)
    m += bucket_size_totals[i];
```

Parallel bucket sort is implemented in both models by recalling their respective libraries. Complete program for P2P-MPI and MPJ-Express can be found at https://github. com/saqibd ola/MulticoreModelsJava. Total number of lines for sequential and parallel bucket sort in P2P and MPJ-Express is shown in **(Table 1).**

Table 1: Total number of lines

Sequential Program	P2P-MPI Program	MPJ-Express Program
38	600	600

4. RESUTS AND DISCUSSIONS

Multiple implementations of bucket sort have been tested for the comparison of MPJ-Express and P2P-MPI. For testing, speedup and efficiency of bucket sort in each model is evaluated. Testing was performed on Core i7 system with 8GB RAM and 6MB cache. MPJ-Express and P2P-MPI programs compilation and evaluation is done in Windows 8.1. Software versions that were used include Microsoft Visual Studio 2012 and Eclipse Debugger.

For performance comparison, we first observed the execution time of sequential bucket program and parallel programs for sorting large number of elements. To ensure the consistency, program was rerun 10 times and average execution time is considered. Table 2 shows the execution time of sequential bucket sort program and execution time of bucket sort in P2P-MPI and MPJ-Express for 3 dataset that was generated randomly. In **(Table 2)**, P denotes the Processors.

Execution time for three programs was measured for three dataset: 8 Million (M), 30M and 100 M. Execution time for parallel programs decreases drastically when compared to sequential program. Execution time for parallel bucket sort program in MPJ-Express was less as compared to the program in P2P-MPI when executed on different number of processors. We also checked the pattern for execution time in both models by running the program in each model with 4 processors. 10 data sets were created to sort elements from 10 million to 100 million using step 10 million. To ensure the consistency, program was rerun 10 times and average execution time is considered. **(Fig. 1)** shows the execution time of parallel programs when executed on 4 processors.

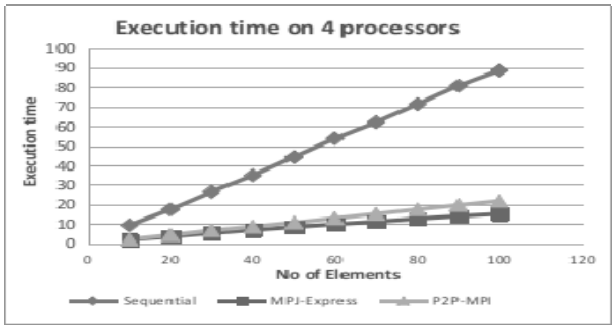


Fig. 1: Execution time for sequential and parallel bucket sort programs

Parallel bucket sort in MPJ-Express has less execution time for sorting elements as compared to bucket sort program in P2P. Beside execution time, we have compared MPJ-Express and P2P for following two parameters:

Speedup: Time a program takes to complete its execution. We have calculated speedup with formula: . T_s represents the execution time for serial program and T_p represents the execution time for parallel one.

Efficiency: How efficiently the processors are being used. Efficiency is calculated as: ., where *p* represents the total number of processors being used.

MPJ-Express and P2P-MPI measures the predicted speed for parallel program on number of processors such as 2, 4, 8 and 16 processors. Fig. 2 shows the speed achieved by parallel bucket sort for dataset 8 M on different processors. X-axis displays the number of processors and Y-axis shows the time taken by each processor.

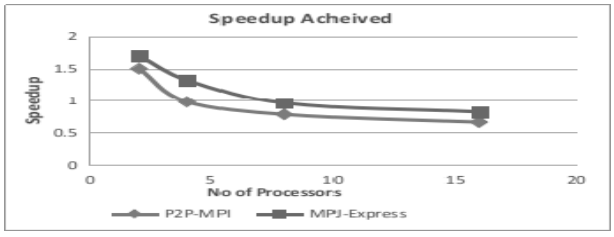


Fig. 2: Estimated speedup achieved by parallel bucket sort

For different number of processors, speedup achieved in MPJ-Express was higher than P2P-MPI. The speedup and efficiency achieved for three dataset when parallel program is executed on 4 processors is listed in **(Table 3).**

Average speedup achieved in both models is shown in **(Fig 3).** Average estimated speedup that is achieved in MPJ-Express was 6.26 and 4.15 in P2P-MPI. In MPJ-Express, approximately 35% high speed and efficiency was achieved as compared to P2P-MPI.

Table 3: Estimated speedup and efficiency for 3 datasets.

No of P	Dataset	Speedup MPJ-Express	Speedup P2P-MPI	Efficiency MPJ-Express	EfficiencyP 2P-MPI
2	8 M	5.06	4.47	2.53	2.23
4	8 M	7.75	5.75	1.93	1.43
8	8 M	9.74	7.91	1.21	0.98
2	30 M	6.1	4.06	3.05	2.03
4	30 M	7.625	4.50	1.90	1.12
8	30 M	8.87	5.570	1.10	0.69
2	100 M	4.02	2.87	2.01	1.14
4	100 M	4.638	3.17	1.092	0.79
8	100 M	8.87	3.25	1.10	0.406

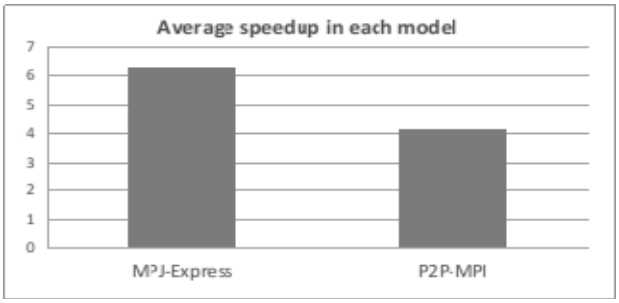


Fig. 3: Average speedup

In general, P2P-MPI took more time for sorting elements when executed on large dataset with less number of processors. Whereas MPJ-Express performed better in each circumstances when executed with any number of processors. We encountered a problem in P2P-MPI when program was executed for very large data size. MPJ-Express easily execute large dataset on any number of processors, but P2P-MPI does not execute large dataset size (more than 100 M) on 8 and 16 processors. However, in P2P-MPI, there is no need for OS binaries as it executes *java.class* file and it can performs self-configuration of frameworks for execution of programs. P2P-MPI also enables fault tolerance for applications that covers the robustness of framework executing program. Processes can be replicated transparently as user configures the number of replicas. On the other hand, MPJ-Express provides thread safety to maintain compatibility with Java threads. MPJ-Express also maintains challenging issues of high performance and portability.

5. COCNUSION

Two parallel programming models P2P-MPI and MPJ-Express for Java are compared by implementing bucket sort algorithm in each model. Comparison was carried out in terms of execution time, speedup and efficiency achieved in each model. Our results showed that MPJ-Express performed better than P2P-MPI for parallel bucket sort program. Generally each multi-core programming model has different pros and cons. Different factors should be considered in order to select the preeminent model in which usability, complexity and feasibility provided by the model are the major aspects to be considered. P2P-MPI and MPJ-Express provide ease of compression proceeding in different perspectives. We come across the notion that in contrast to MPJ-Express, P2P-MPI is out of scope for the scrutiny to be performed here. MPJ-Express is valuable to be used for educational practice and for other ranges of concurrency. MPJ-Express is better for parallelizing the programs that require greater speed as it is more flexible than P2P-MPI. In future, one interesting area is to compare the performance of MPJ-Express and P2P-MPII with their respective implementations in GCC and in other compilers as well in order to investigate the consistency of the results presented in this work across different implementations.

REFERENCES:

Chandra, R., L. Dagun, D. Maydan, J. McDonald, and R. Memon (2001). Parallel programming in OpenMP. USA: Academic Press. USA, 6-28.

Gropp, W., E. Lusk and A. Sjkellum (1999). Using MPI: Portable parallel programming with the message-passing interface. MIT Press, USA.

Hayes, B (2007). Computing Science: Computing in a parallel universe. American Scientist, 95(6):476.

Jarp, S., A. Lazzaro, A. Nowak, and L. Valsan (2012). Comparison of software technologies for vectorization and parallelization. CERN OpenLab, Technical Report.

Kessler, C., and J. Keller (2007). Models for parallel computing: Review and perspectives. PARS Mitteilungen, 24:13-29.

Krieger, C., M. Strout (2012) Executing optimized irregular applications using task graphs within existing parallel models. In Proc. of High Performance Computing, Networking, Storage and Analysis, 86-93.

Liest, A, and and A. Gilman (2014). A comparative analysis of parallel programmingmodels for C In Proc. of 9th International Multi-Conference on Computing in the Global Information Technology, 121-127.

Nichols, B., D. Buttlar and J. P. Farrell (1996). Pthreads programming: A POXIS standard for better multiprocessing. USA: O'Reilly, Inc. USA, 15-28.

Nichols, B., D. Buttlar and J. P. Farrell (1996). Pthreads programming: A POXIS standard for better multiprocessing. USA: O'Reilly, Inc. USA, 15-28.

Nanz, S., S. West, K. S. Silveria and B. Meyer (2013). Benchmarking usability and performance of multicore languages. In Porc. of ACM/IEEE International Symposium, 183-192, 2013.

Nawaz, M. S., M. I. Lali, S. Saleem and H. Ali (2014). Comparison of multi-core parallel programming models for divide and conquer algorithms. Ned University Journal of Research, 8:1-8.

Podobas, A., and K. F. Faxn (2010). A comparison of some recent task-based parallel programming models. In Proc. of 3rd Workshop on Programmability Issues for Multi-Core Computers, pp. 12-19.

Robison, D. A (2013). Composable parallel patterns with Intel Cilk Plus. Computer in Science and Engineering, 15(2): 66-71.

Reinders, J (2007). Intel threading building blocks: Outfitting C for multi-core processor parallelism. USA: O'Reilly Media, Inc. USA, 215-223.

Saule, E., and U. Catalyurek (2012). An early evaluation of the scalability of graph algorithms on the Intel MIC architecture. In Proc. of Symposium of Parallel and Distributed Processing, 1629–1639.

Saleem, S., M. I. Lali and M. S. Nawaz (2014). multi-core program optimization: parallel sorting algorithms in Intel Cilk Plus. International Journal of Hybrid Information Technology, 7(2): 151-164.

Silva, L. M., and Buyya (1999). Parallel programming models and paradigms. High Performance Cluster Computing: Architectures and Systems, 2(3): 4-27.

Tristram, W., and K. Bradshaw (2010). Investigating the performance and code characteristics of three parallel programming models for C++. In Proc. of Southern Africa Telecommunication Networks and Applications Conference, 122-129.

Taboada, G. L., S. Ramos, R. R. Exposito, J. Tourino and R. Doallo (2013). Java in the high performance computing arena: Research, practice and experience. Science of Computer Programming, 78(5): 425-444.

Wahlen, N (2010). A Comparison of different parallel programming models for multicore processors. *Masters Thesis*, KTH Information and Communication Technology.

Zafar, H., F. A. Khan, B. Carpenter, A. Shafi and W. Malik (2015). MPJ Express meets YARN: Towards Java HPC on hadoop systems. Procedia Computer Science, 51: 2678-82.