A Parallelisation Tale of Two Languages

Antonio J. Tomeu[1], Alberto G. Salguero[1], and Manuel I. Capel[2]

1 Department of Computer Science
University of Cádiz
Spain
antonio.tomeu@uca.es, alberto.salguero@uca.es

2 Department of Computer Languages and Systems
University of Granada
Spain
manuelcapel@ugr.es

Abstract

Since there are nowadays several APIs (C+11, TBB, OpenMPI, . . .) that properly support codeparallelism in any of the most used programming languages (C++, C#,Java, Phyton, or Erlang) in telecommunications industry, for a novel programmer of multicores, it may become very difficult to know how to choose the appropriate programming language to get their sequential code parallelised well up to N processors.
C++11 is the new ISO standard for C++, which offers atomic declaration of variables,load/store operators to programmers and supports portable multithreaded programming in this familiy of development languages.
Since version 5.0, Java SE has been including new primitives for doing high–level concurrency in Java programs. Currently, Java SE 8 also offers parallel syntactical constructs for new multicore architectures programs, such as lambda functions with an efficient tail recursion implementation, execution to future, etc.
Within a similar historical context, as in the Charles Dicken's history of 2 cities, there may exist equivalent problems in two close–up places but they can be solved in a quite different way. The main motivation of this work is the next one: we believe that there still little fundamentalresearch done on how they compare different modern concurrent programming languages and concurrency APIs and on how they differently implement similar parallel and syntactical constructs. Moreover, among the considered more competitive concurrent programming languages today, to determine which one is offering better results regarding shorter execution time and performance when writing parallel for multicore architectures is still a pending issue.
We perform a systematic comparative study between 2 programming languages that have become increasingly deployed in industry over recent years, probably due to their popular parallel programming APIs Java Concurrency Utilities and C++11, respectively. In this respect the paper describes how to test real capabilities of Java SE 8 and C++11 to accelerate a certain class of numerical calculations depending on several factors (locality, subproblems granularity and number of parallel tasks). Solutions to tackled problems have been programmed with no race conditions or latencies associated with locks. Therefore, we are applying the same algorithmic scheme to obtain a parallel calculation based on a well-known discrete model that uses cellular automata to solve two study–case problems and their solutions, the Belousov-Zhabotinsky's chemical reaction and a square matrices convolution calculus. We carry out a set measures to compare the execution time and performance (parallel speed-up) if we perform measurements that exclude overheads due to thread creation and launching, which could make different parallel deployment and concurrent management of thread models in both languages.
Differently, from other similar works on the subject, we focus more on foretelling performance results that different APIs are offering to non-specialized programmers for a specific piece of code or algorithmic problem class than in comparing the low–level implementation efficiency of syntactical constructs in the modern concurrency APIs.

© Antonio J. Tomeu, Alberto G. Salguero and Manuel I. Capel;
licensed under Creative Commons License CC-BY
Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1998 ACM Subject Classification D.3.2 Concurrent, distributed, and parallel languages

Keywords and phrases High-level concurrency API, performance comparison, multicore programming, parallel programming, simulation of physical phenomena, Belousov-Zhabotinsky's reaction, parallel 2D convolution.

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

For years, the direction followed in the evolution of processors has been aimed at increasing the number of cores or processing units, which not only gives us the ability to parallelize tasks on personal computers, but also to use those resources for calculations using parallel computing. Parallel programming has traditionally relied on proprietary tools for application development, including classic programming languages, direct control of parallelism using specific purpose sets of directives for this objetive. MPI and OpenMP ([2]) are classic examples of this, and both of them have bindings for languages like C, C++ and Fortran.
Today, multithreaded programming is not one choice that applications developer may or may not be aware; it is a compulsory way to follow, considering the technological reality that any modern multicore platform offers. In last decades there has been an intensive research on comparisons between different models of parallelism: System V processes (*fork/join*), multithreading, tasks, MPI, etc.
Yet there is still little research to compare how different current programming languages implements each different concurrent and parallel constructions, and wich are considered more competitive concerning their performance (low runtime programs).
Moreover, the aforementioned growing presence of multicore processors in all areas of software development, has required to many languages the integration into their APIs the ability to create and control threads in a more natural and simple way to the programmer ([10], [11], [19] and [25]). The Java language, from its first specification, gives the programmer the ability to have concurrent threads by two different techniques: inheritance by extending the *Thread* class or implementing the *Runnable* interface. From 1.5 specification, are further incorporated the *Callable* interface, in combination with *Future* interface allows asynchronous computing tasks further provide a result ([11] and [19]) will not immediately obtainable.
By contrast, the dyad C/C ++ has not had capacities to create and manage concurrent tasks integrated into the language API to the last stable release (2011) (the review "minor" in 2014 has added nothing new in this field). Previously, programmers have been used for the development of concurrent sotfware two approaches:

- Concurrency with processes using the fork system call; approach that can be considered obsolete, as does not create lightweight processes (threads), and also requires the use of external libraries to make the processes share memory or can be synchronized. This is achieved through the facilities of the System V IPC specification and, more recently, POSIX IPC ([22]). To synchronize processes using semaphores, you need to make an explicit connection to shared memory regions previously created, and a final disconnection, not allowing high-level support for writing monitors according to the Hoare's semantics and integrates wrong with orientation objects.
- Concurrency with threads, using the IEEE POSIX 1003.1c, as embodied in the use of *phtread.h* library, providing the programmer facilities for the creation and management of threads, basic locks to control synchronization, and condition variables ([22]), which

may introduce race conditions between threads.

From the ISO/IEC 14882: 2011 review (we insist: the modification of 2014 does not incorporate any modifications to this) the C++ language has included ([8]) a set of capabilities into the API for the creation and control of concurrent threads that can be homologous to other general purpose languages ([25]) and concurrent object-oriented ([8] and [25]).
Therefore, a research topic of undoubted interest at present is to verify if the performance runtime for the same programs written with C/C++ (not using support virtual machine) improves performance obtained using languages with a virtual machine as support running (Java, C#) to achieve neutrality of architectures. To test this hypothesis we had made a comparative study based on simple performance benchmarks using (only) C++11 and Java v/8.0 profiles as more representative of imperative languages and managed by a virtual machine respectively. To do the study we have choose two problems using big collections of numerical data:

- Simulation with cellular automata of the Belousov-Zhabotinsky chemical reaction.
- Discrete 2D-Convolution.

Comparative will be developed in the following sections. Section II discusses the tools to have concurrent/parallel tasks in both languages; section III focuses on to simulate with cellular automat the Belousov-Zhabotinksy chemical reaction, calculating speed up and time as function of number of parallel tasks; section IV do the same one for discrete 2D convolution; section V discusses the comparison of results between the two languages and Section VI provides some conclusions and sets future work.

2 C++11 vs. Java Concurrent/Parallel Task Creation

2.1 Creating and Modeling Concurrent/Parallel Tasks

New C++ standar defines the thread class, allowing the programmer to instantiate threads comfortably. There are two techniques ([25]) to send a thread the segment of code that it should process:

- Wrapping the code expressing concurrency in a function that constructor takes as argument with a parameter that is a function pointer.
- Using a function lambda that allows wrap that code directly, without resorting to a reference function that contains the code. This second technique is often used ([25]) when the code is passed to the thread to be processed is very short.

Now, we show examples for first

```
#include <thread>
using namespace std;
void thread_code(){cout<<"Hello"<<endl;}
int main(){
  thread t(thread_code);
  t.join();
}
```

and second technique

```
#include <iostream>
#include <thread>
```

```
#include <vector>
using namespace std;

int main(){
  vector<thread> threads;
  int nthreads = 100;
  for(int i=0; i<nthreads; ++i)
    {threads.push_back(thread([](){cout <<"Hello"
             << this_thread::get_id()<< " ";}));}
  for(auto& thread : hilos){thread.join();}
  return(0);
}
```

Java also uses two techniques ([14], [15] and [19]) of threads creation, the first of which is specific to this language and does not have approval to C++, while the latter is conceptually similar to the use of function pointer in C++:

- Inheritance by class extension. The programmer inherits from the Thread class and overrides *run()* method to equip the new class objects (which are already threads) the behavior he wants. This technique has a serious limitation, since the extension of class inheritance in Java is not manifold, and therefore the programmer is unable to inherit from another class if needed.
- Implementing the Runnable interface, where the programmer implements this interface in a new class that gives meaning to the calculation to be performed by the thread through the implementation of *run()* method of this interface. The new class thus obtained is homologous to the function using C++ and, if applicable, is not a thread, but a parameter to the constructor of the Thread class should receive. This technique circumvents the problem of exhausting the possibilities of extending the class that occurs in the previous case, as the new need not extend Thread, and therefore is used habitually.

Now, we show examples for first

```
public class MyThread extends Thread{
  public MyThread(){}
  public void run (){
    System.out.println("Hello");
  }
  public static void main (String[] args){
    MyThread t1 = new MyThread();
    t1.start();
    t1.join()
  }
}
```

and second technique

```
public class MyRunnable implements Runnable{
public MyRunnable(){}
public void run(){
  System.out.println("Hello");
}
public static void main(String[] args) {
  MyRunnable m1 = new MyRunnable();
  Thread t1=new Thread(m1);
  t1.start();
  t1.join();
 }
}
```

Both languages also use computations to future of asynchronous tasks. If a thread needs to wait for a specific event, obtains a representation future of it. The thread can then periodically check if the event occurred, and even get the result of a calculation. But thread can also do more but wait the result, in the interval between checks. In the case of C++11 language, there are two templates ([25]) declared in the header *Future*⟨*type*⟩ to manage this model.Meanwhile, from version 1.5 Java models this concept depicting asynchronous tasks by *Callable* interface; the programmer implements it according to their needs and using *Future*⟨*type*⟩ interface too. The future can check the status of *Callable* objects and, when available, get the result it produces.

2.2 Concurrent Task Management

We show in this point a quick summary of issues related to the management of concurrent tasks in both languages:

- Execution: C++11 schedules to run a thread right away the object thread is created, whereas Java requires an explicit signal to the Java virtual machine (JVM) by the *start()* method, causing an already created thread could be scheduled for execution.
- Both languages allow to have the necessary control for the main thread waits for the purpose of executing a set of daughter threads methods using type *join()* methods.
- Equally both languages provide the programmer with comparable semantics, methods to pause a thread for a specified time (sleep ()).
- Also available in both languages threads running in the background (daemon threads), normally of long duration, and performing tasks of supervision or maintenance throughout the entire lifecycle of the application. To have them available in C++ the *detach ()* method on a thread already created is used, whereas Java uses the *setDaemon(boolean on)* method of the Thread class.

3 Belousov-Zhabotinsky's Chemical Reaction Simulation

Cellular Automata have been used to develop discrete models for some chemical reactions ([9]) and to simulate physical phenomena of real world ([7]). Belousov-Zhabotinsky's (BZ) chemical reaction is one of them. In this paper, we will use an explicit reaction model to simulate BZ reaction, ir order to parallelize it. Cellular automata, at a basic level, may be defined as a collection of cells that may be in one state of several (finite) possible. The system evolves in discrete time by any law (the transition function) that modifies the state of each cell to a new one, depending on the previous state and its neighborhood of cells.The model we propose use set states for representing quantities of three chemical substrates by floating point values. The method is similar to a reaction-diffusion model ([1]), using a summation of states surrounding cells and of course, taking discrete time steps. You can see the real evolution of the BZ reaction on a *Petri* dish in Figure 1.

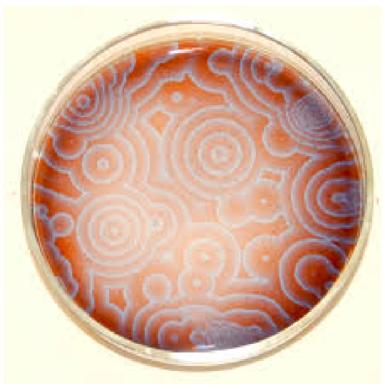


Figure 1 BZ Chemical Reaction within a Petri dish.

Ball ([5]) makes a description of the BZ reaction as a series os chemical equations to the form $A + B \rightarrow 2A$. What may mean this equation? Reallys is very simple: a quantity of B is provided and the creation of A is autocatalyzed until the supply of B expires. We can form sets of competing reactions adding similar chemical equations:

$$B + C \rightarrow 2B \tag{1}$$
$$C + A \rightarrow 2C \tag{2}$$

These equations are not exactly as formulated by *Ball*. The circularity is very useful to create simple implementations. With these new equations, B can be created, but only if there is a quantity of C and finally, C can be created, but only is there is a quantity of A, bringing the reaction full circle. That set of three equations will be used for our basic simulation using cellular automaton in this paper. With the reactions described, equations may be written for the quantity of A, B or C present as a function of time. The symbol a_t will be used to mean the quantity A at time t, similarly, b_t will denote the quantity of B at time t and c_t the quantity of C. In these symbols, the quantities of A, B and C at time $t+1$ be written as:

$$a_{t+1} = a_t + a_t(b_t - c_t) \tag{3}$$
$$b_{t+1} = b_t + b_t(c_t - a_t) \tag{4}$$
$$c_{t+1} = c_t + c_t(a_t - b_t) \tag{5}$$

We can see how in time $t+1$ each quantity is dependent on two competing processes. The quantity of A at $t+1$ is increased according to the quantity of B present, but decreases due the quantity of C, as A is used to create C. Sometimes, additional parameters can be added in order to change the reaction rates of the competing processes. We do no add new parameters here, for the purposes of clarity. With these equations, it is possible use them to set up an oscillating reactions at a single location, but we want to create a diffusion surface with two dimensions. To do it, wi will employ a cellular automata model. With this kind of discrete models, the amount of A, B and C is simply averaged for each cell and their neighborhood before the reaction equations above are applied[1]. The new value for the cell at position

[1] That is, the values of a_t, b_t y c_t are simply summed for the 9 cells in the immediate vicinity (including

(x, y) in then calculated inserting the average values into the reaction equations above. This could clearly be enhanced to take into account the previous quantity of each substrate at the location, rather than its average, but it is satisfactory for this implementation. Now, we propose an algorithm to simulate BZ reaction using a cellular automata. To do it, we have added to below equations some parameters in orden to can modify the wavefront of the reaction. This wavefront usually appears after a few generations and stabilized around hundred of them. Adjusting the relative rates of the reactions the wavefront will change. New equations are:

$$a_{t+1} = a_t + a_t(\alpha b_t - \gamma c_t) \quad (6)$$
$$b_{t+1} = b_t + b_t(\beta c_t - \alpha a_t) \quad (7)$$
$$c_{t+1} = c_t + c_t(\gamma a_t - \beta b_t) \quad (8)$$

With the above equations, we can derive the desired algorithmic simulation, in single thread version:

```
float [][][] a;
float [][][] b;
float [][][] c;

int p      = 0;
int q      = 1;
int width  = 1600;
int height = 1600;
alfa       = 1.2f;
beta       = 1.0f;
gamma      = 1.0f

void setup (){
  a = new float [width][height][2];
  b = new float [width][height][2];
  c = new float [width][height][2];

  for (int x = 0; x < width ; x ++) {
    for (int y = 0; y < height ; y ++) {
      a[x][y][p] = random (0.0 ,1.0);
      b[x][y][p] = random (0.0 ,1.0);
      c[x][y][p] = random (0.0 ,1.0);
    }
  }
}

void compute (){
  for (int x = 0; x < width ; x++){
    for (int y = 0; y < height ; y++){
```

the central cell), and divided through by 9.

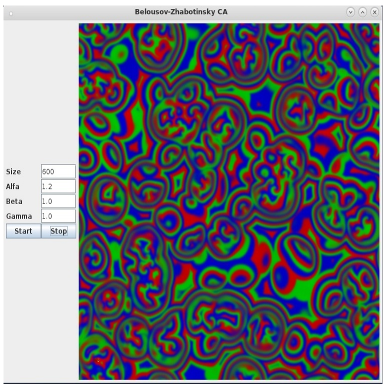
```
      float c_a = 0.0;
      float c_b = 0.0;
      float c_c = 0.0;
    for (int i = x-1; i <= x+1; i++){
      for (int j = y - 1; j <= y +1; j++) {
        c_a += a[(i+ width)%width][(j+height)%height][p];
        c_b += b[(i+ width)%width][(j+height)%height][p];
        c_c += c[(i+ width)%width][(j+height)%height][p];
      }
    }
    c_a /= 9.0;
    c_b /= 9.0;
    c_c /= 9.0;
    a[x][y][q] = constrain(c_a+c_a*(alfa*c_b-gamma*c_c));
    b[x][y][q] = constrain(c_b+c_b*(beta*c_c-alfa*c_a));
    c[x][y][q] = constrain(c_c+c_c*(gamma*c_a-beta*c_b));

  }
  if(p==0){p = 1; q = 0;}
    else {p = 0; q = 1;}
 }
}
```

As seen, we have used Moore's neighborhood and cylindrical boundary conditions, so simulation will happen on a toroidal structure ([13]). We also clarify that the values are calculated and constrained to the range $[0, 1]$. This restriction is only established so that the values stay in a range for color display purposes (see Figure 2). The equations as stated maintain a equal amount of substrate overall, although individual cells might overstep the level. The algorithm alternates the values of p and q as the current time step and the next time step so that values do not have to be explicitly transferred from step $t+1$ to step t, enabling parallel reading and writing data. Once new values for the level of each substrate have been updated, the only remaining line is to set the color of each pixel according to the level of the substrate. Emulation of an indicator is chosen, so that the amount of color is dependent on the amount of substrate A at any one location. This could be made more colorful by adapting it so that it works of hue rather than brightness.

From below code, we have made two single thread implementations using C++ and Java languages. In both cases, and for a 1600×1600 matrix of data, simulation is very hard in time to do, because only one core of the processor in being employed. From here, we are going to develop a parallel version on the below algorithm, deriving implementations in C++ and Java.

Next step will be make a theoretical parallelisation of `compute()` function. We choose a very simple scheme ([24]) for data partition. Suppose we have four cores available in our machine; data partition will assign four 400×1600 matrix to four different task. Each task will run on a different thread with a dedicated core. No locks are necessary, because we are using two arrays of data for actual and next state of cells. A condition of synchronization



Size 600
Alfa 1.2
Beta 1.0
Gamma 1.0

Figure 2 Simulation of BZ Chemical Reaction with Celullar Automata.

between the threads to move from one state[2] to the next one will be required. The `compute()` function was changed by the next one:

```
void compute (int width_low, int width_high)
```

Now, the main program calls several times (in parallel) the function `compute()` embedded within a thread, and sends it two parameters with information about how to locate the segment matrix on which to work. Parallels tasks work with the same data matrix, and compute next generation in an auxiliar matrix, so we can to have parallel reads and writes. While each task performed his part of the work, the main program wait all of them, and then repeat the process.

4 2D-Convolution

Convolution is a widely used technique in image and signal processing applications. In image processing the convolution operator is used as a filter ([20]) to change the characteristics of the image; sharpen the edges, blur the image or remove the high or low frequency noise. In seismic processing ([26]) a convolution can be used to extrapolate the propagating wavefield forward of backward. In signal processing it can be used to suppress unwanted portions of the signal or separate the signal in different parts. By its nature, the convolution operation performed on clouds of large numerical data (very big pictures, by example) requires high processing times. Therefore it is interesting to have parallel versions of the convolution available. This section will study how are capable of accelerating the convolutions the C++11 and Java languages, for different number of parallel tasks.

As is known, the two-dimensional convolution of a data matrix can get it from the following equation:

$$g(x,y) = h(x,y) * f(x,y) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} f(x,y)h(x-y,y-j) \tag{9}$$

[2] Once each thread has finished its part of the job.

where f is the original function and g is the resulting convolution function and h is the convolution function, usually called mask or kernel. In our case, we will use as usual 3×3 kernels and the above equation can be rewritten as:

$$g(x, y) = h(x, y) * f(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} f(x, y)h(x - y, y - j) \tag{10}$$

Now, using the last equation and assuming the identity kernel (this is no important here; we can to choose other kernels, and processing times not change significantly) we propose a single thread algorithm to make 2D convolutions:

```
float [][] in, out;
float [][][] kernel=  { { 0, 0, 0},
                        { 0, 1, 0},
                        { 0, 0, 0}
                      };

void setup (){
  heigth = 1600; witdth = 1600;
  in     = new float [width][height];
  out    = new float [width][height];

  for (int x = 0; x < width ; x ++)
    for (int y = 0; y < height ; y ++) {
      in[x][y]  = random (0.0 ,1.0);
      out[x][y] = 0.0f;
    }
}

void convolution (){
  for (int x = 0; x < width ; x++)
    for (int y = 0; y < height ; y++)
      for (int i = x-1; i <= x+1; i++)
        for (int j = y - 1; j <= y +1; j++)
          out[x][y] += out[x][y]+in[(i+width)% width][(j+height)% height]
             *kernel[i][j];
}
```

Again, we have used cylindrical boundary conditions, so simulation will happen on a toroidal structure and this time, our kernel is the identity although our software lets you choose from a wide variety of filters. Notwithstanding the foregoing, it is necessary to make a clarification: aspects of convolution as image processing technique, are completely irrelevant here. The interest of their choice as a case study takes the form of a problem that uses large amounts of numerical data, and whose parallelisation is free of latencies due to locks or other synchronization techniques. Moreover, the proposed algorithm to calculate convolutions is elemental, and requires no further explanations.

Next step will be make a theoretical parallelisation of `convolution()` function. We choose a very simple scheme for data partition ([4]), as we did with BZ case. Suppose we

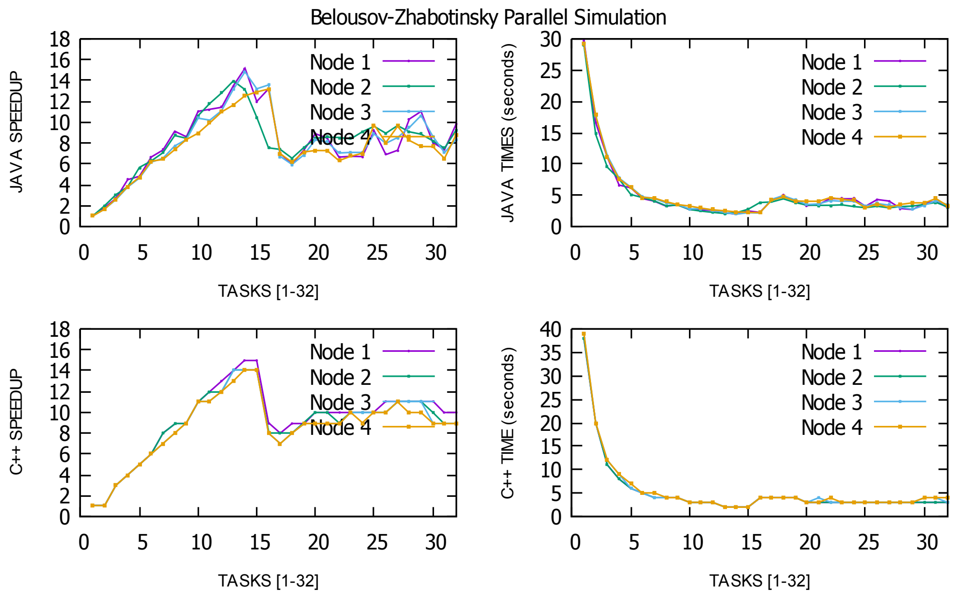


Figure 3 Speed Up/Execution Time for BZ Parallel Simulation

have four cores available in our machine; data partition will assign four 400 × 1600 matrix to four different task. Each task will run on a different thread on a single core. This time, no locks are necessary, because we are using two arrays of data (`int`, `out`) for original data and convolved data. Again, a condition of synchronization between the threads to move from one state[3] to the next one will be required. The `compute()` functions was changed by the next one:

```
void convolution (int width_low, int width_high)
```

Now, the main program calls the function `compute()` embedded within a thread, and sends it two parameters with information to locate the segment matrix on which to work. Parallels tasks work with the same data matrix, and compute next generation in an auxiliary matrix, so we can to have parallel reads and writes. While each task performed his part of the work, the main program wait all of them, and then repeat the process.

5 Results

Parallel simulations of BZ and convolution was runned on four nodes of our university cluster. Each node has two processors Intel Xeon[4] E5 2670, with 2.6 GHz. of clock frecuency, 128 GB. of RAM and 20 MB. of smart cache (L3), amounting to a total of 16 physical cores available. All nodes run Enterprise Linux Input 6.4 operating system; input nodes run Server with High Availability version and compute nodes run Compute Node one. The management of compute nodes is made by CMU (Cluster Management Utility) by Hewlett-Packard. Initial data matrix was preloaded with random data. Compilers used were:

- Suns's SDK version 1.7.0_40.

[3] Once each thread has finished its part of the job.
[4] In these machines, hyperthreading is not activated.

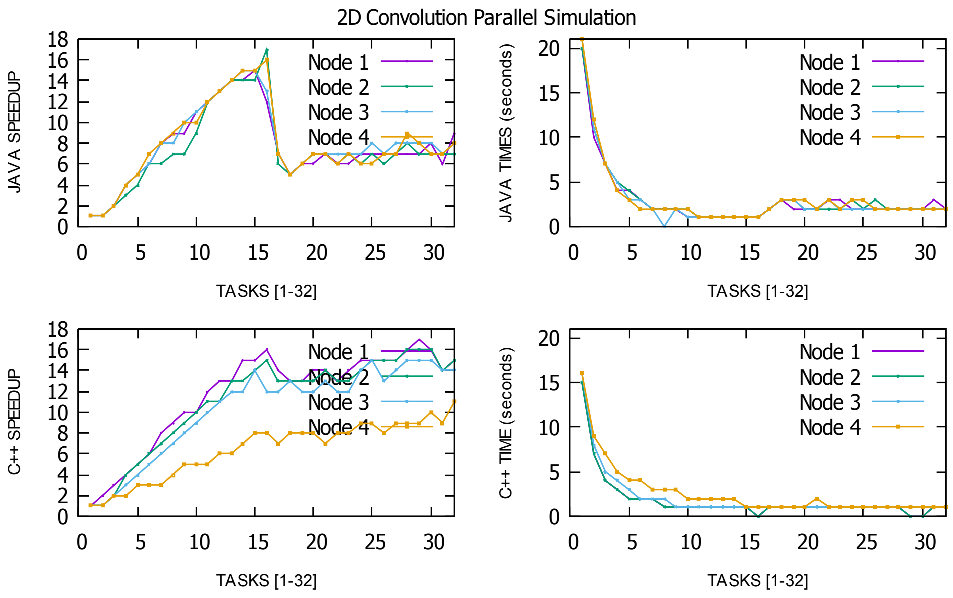


Figure 4 Speed Up/Execution Time for 2D Convolution Parallel Simulation

- GNU's GCC 4.4.7.

Figure 3 first column shows speed ups as a function of the number of tasks, for Java and C+ + 11 languages, doing the simulation in four nodes and on 1600 × 1600 matrix of elements, computing the evolution of the reaction for 100 generations. Data for each node are showed. We can see how the speed up obtained by parallel simulation grows as we increase the number of tasks. The simulation provides the best performance when we have a task per core as expected ([23], [11] and[12]). From here, we see the increasing number of parallel tasks worsens speed up, which is stabilized in a plateau. We see that C++11 provides values of speed up slightly better than Java, which had already been proven by other authors ([12]). Both C++11 as Java offer hyperlinear speed up for a number of parallel tasks equal to or less than the number of cores. Hyperlinearity in this computer can be justified (as in [6], [16], [3], [18] and [21]) for the type of problem, since it has a core task that runs entirely in parallel with others without locks of any kind; furthermore, calculations done during simulation to recalculate the value of each cell of the cellular automaton requires the immediate neighboring data (remember we are using Moore's neighborhood) and therefore the data locality is very high.

Figure 3 second column shows running times as a function of the number of tasks, for Java and C++11 languages. Now, the best time of parallel processing is achieved with a task running per physical core. To scale the number of tasks beyond 16 does not provide additional benefits. Is interesting to note here that Java gets better computation times than C++11 generally. This behavior, which had already been observed by other authors ([12]) can be explained by way of processing threads used into Java implementation: a thread pool is employed to manage lifecycle of tasks. This technology when we used in multithreaded Java programs can achieve significant improvements in runtime ([11] and [23]).

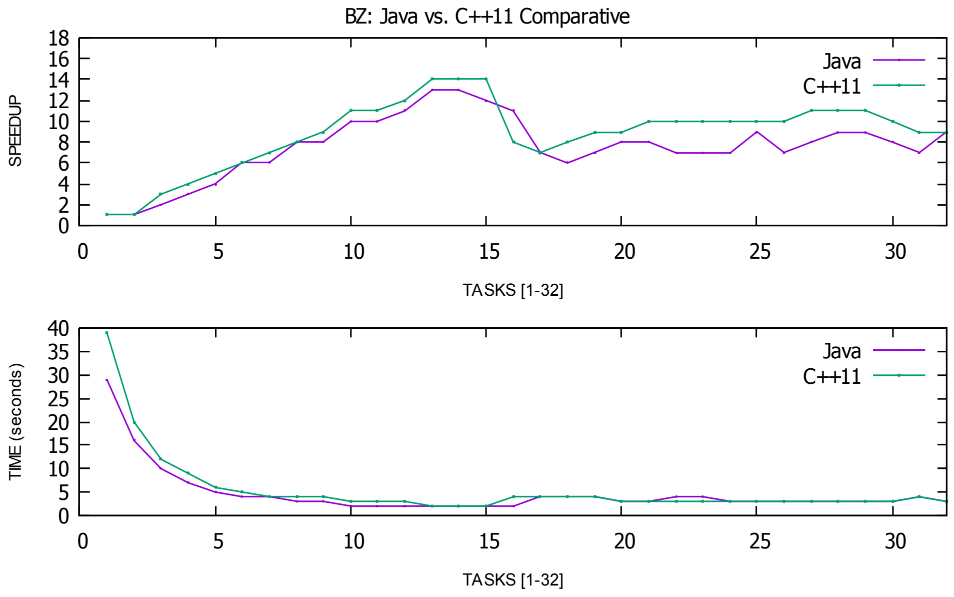


Figure 5 Java vs. C++11 in Parallel BZ

Results for 2D Convolution were very similar, and whe show then in Figure 4. With convolution, C++11 offers better speed ups and runtime than Java. Hyperlinearity phenomena were also observed, and for values slightly higher than in BZ. Recall that convolution makes a single processing pass over data data arrays while BZ makes a hundred, and that convolution requires two data arrays while BZ requires five.

Comparative analysis of both languages for BZ parallel simulation is shown in Figure 5. We can see how speedup is slightly better when we use C++11 for a number of tasks smaller than physical cores number. Beyond, the trend remains and C++11 performs better. Moreover, an analysis of the execution times reveals a very similar behavior. A very slight advantage of Java over C++11 ([12]) below 16 tasks, and virtually identical behavior for more than 16 tasks. As expected ([11] and [23]), and for this type of problems, a number of tasks higher than cores does dramatically worsen the behavior of both variables: speed up and runtime. This is due to the increased overhead.

Finally, we can see the comparative analysis for parallel 2D Convolution in Figure 6. In terms of speedup and qualitatively, we obtain a performance equal to the previous case. Quantitatively changes are observed, and we can see how the gap between Java and C++11 is greater than for BZ. This is not surprising: for BZ parallel simulation, a hundred of generations were carried out on data arraus, in which the life cycle of the tasks was managed by a thread pool (which is not currently available in C++); thread pool enabled Java makes a more efficient management of this cycle reusing threads into the pool, and have a lower disadvantage compared with C++ 11. For runtimes, unlike BZ, Java behaves worse than C++11 in the whole tasks domain. Again, the reason for this is attributable to the presence of a thread pool within Java implementation. While BZ performs a hundred of iterations on data (and this allows the pool to improve the runtime), convolution makes a one-step calculation on data arrays. Create a thread pool is very hard in terms of runtime, and Java must pay the price.

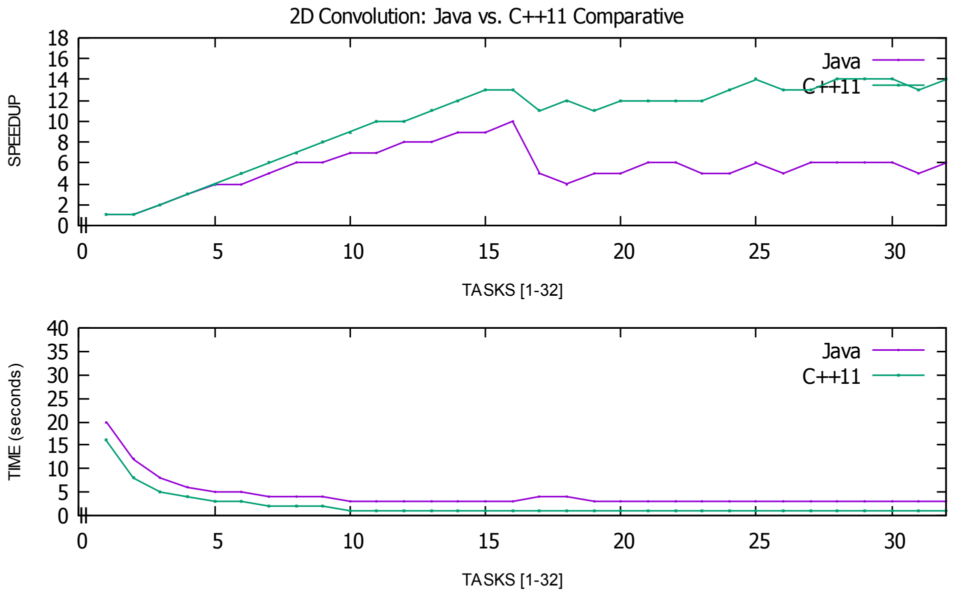


Figure 6 Java vs. C++11 in Parallel 2D Convolution

6 Conclusions and Future Work

Parallel implementations of BZ and convolution is proposed. Fist implementation is made approaching the theoretical model of the reaction by a cellular automaton. 2D Convolution implementation is straightforward and standard. For both problems parallel implementations in Java and C++11 was written, and a comparative analysis of speed up an time was made. Results showed that for problems with relatively short processing times C++11 always behaves better than Java for two variables analyzed. However, calculations for BZ showed that for higher data clouds requiring higher processing times, Jave language offers better runtimes C++11, although worse speed up. Our future work will focus on extending the analysis presented here with bigger data clouds, a typology of broader problems, and a third variable of analysis: the granularity of the data.

References

1 Adamatzky, A., De Lacy, B. & Asai, .T. Reaction-Diffusion Computers. Elsevier B. V., 2005.

2 Akhter, S.& Roberts, J. Multicore Programming Incresing Performance Through Sofware Multithreading. Intel Press, Digital Edition, 2006.

3 Alba, E. Parallel evolutionary algorithms can achieve super-linear performance. Information Processing Letters 82, 7-13, 2002.

4 Al Umairy, S., Van Amesfoort, A., Setija, I., Van Beurden, M. & Sips, H. On the Use of Small 2D Convolutions on GPUs. Lecture Notes in Computer Science, volume 6161, 52-64, 2012.

5 Ball, P. Designing the Molecular World: Chemistry at the frontier. Princeton University Press, 1994.

6 Bai, Q., Shao, Y., Pan, D., Zhang, Y. Liu, H. & Yao X. Parallel high-performance grid computing: capabilities and opportunities of a novel demanding service and business class

allowing highest resource efficiency. Studies in health technology and informatics, 01/2010, 159-264-71, 2010.

7 Bandman, O. Mapping Physical Phemomen onto CA-Models. Automata-2008. Theory and Applications of Cellular Automata, 381-395. Luniver Press, 2008.

8 C++ Reference. (`http://en.cppreference.com/w/`). 2014.

9 Deutsch, A. & Dorman, S. Cellular Automaton Modeling of Biological Pattern Formation. Characterizacion, Applications and Analysys. Birkauser Boston, 2005.

10 Fernandez, J. Java 7 Concurrency Cookbook. Packt Publishing, 2012.

11 Göetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. & Lea, D. Java Concurrency in Practice. Addison-Wesley, 2006.

12 Goram. A.K. & From, A. A Comparative analysis between parallale models in C/C++ and C#/Java. (`http://kth.diva-portal.org/smash/get/diva2:648395/FULLTEXT01.pdf`). 2013.

13 Hao W., Lu Z., Xu C., Yan L. & Yurong S. Quantifying and analyzing neighborhood configuration characteristics to cellular automata for land use simulation considering data source error. Earth Science Informatics, Volume 5, Issue 2, 77-86, 2012.

14 Java Platform, Standard Edition 8. API Specification (`http://docs.oracle.com/javase/8/docs/api/`). Oracle Corporation.

15 Lea, D. Programación Concurrente en Java. Principios y Patrones de Diseño. Addison Wesley, 2000.

16 Nagashima U., Hyugaji, S. Sekiguchi, S. Sato, M. & Hosoya, H. An experience with superlinear speedup achieved by parallel computing on a workstation cluster: Parallel calculation of density of states of large scale cyclic polyacenes. Parallel Computing, volume 21, issue 9, 1491-1504, 1995.

17 Mikhailov, A. S. and Showalter, K., Control of waves, patterns and turbulence in chemical systems. Physics Reports 425, 79-194, 2006.

18 Nefedev, K.V. & Peretyako, A.A. Superlinear Speedup of Parallel Calculation of Finite Number Ising Spins Partition Function. Proceedings of Third International Conference of High Performance Computing HPC-UA, 282-286, 2013.

19 Oaks, S. & Wong, H. Java Threads, 3rd Edition. O'Reilly, 2004.

20 Petrou, M & Petrou, C. Image Processing: The Fundamentals. John Wiley & Sons. 2010.

21 Rauber, T. & Rünger, G. Parallel Programming for Multicore and Cluster Systems. Second Edition. Springer-Verlag, 2012.

22 Robbins, K. & Robbins, S. Practical Unix Programming. A Guide to Concurrency, Communication and Multithreading. Prentice Hall, 1996.

23 Subramanian, V. Programming Concurrency on the JVM: Mastering Synchronization, STM and Actors. The Pragmatic Programmers, 2011.

24 Sharifulina, A. & Elokhin, V. Simulation of Heterogeneous Catalytic Reaction by Asynchronous Cellular Automata on Multicomputer. Parallel Computing Technologies. Lectures Notes in Computer Science, volume 6873, 204-209, 2011.

25 Williams, A. C++ Concurrency in Action. Practical Multithreading. Manning, 2012.

26 Wadhawam, M., Midha, P. Kaur, I. & Kaur, S. An Investigación of the Tools of Seismic Data Processing. Proceedings of the 8^{ht} Biennial International Conference & Exposition on Petroleum Geophysics, 303-309, Hyderabad, 2010.