ANN. MULT. GPU PROG.

A Parallelisation Tale of Two Languages

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

*Correspondence:
antonio.tomeu@uca.es
[1]Department of Computer Science, University of Cádiz,Avda. de la Universidad de Cádiz, 10, 11519 Puerto Real, Cáadiz, Spain
Full list of author information is available at the end of the article

Abstract

There are several APIs (C+11, TBB, OpenMPI, ...) that properly support code parallelism in any of the most used programming languages (C++, C#,Java, Phyton, or Erlang) in telecommunications industry nowadays, but it is a complicated issue for a novel programmer of *multicores* to decide which one to choose in order to get an application parallelized with high performance.

C++11 is the new ISO standard for C++, which offers *atomic declaration of variables*, *load/store* operators and supports portable multithreaded programming in this familiy of development languages.

Java SE has been including new primitives for doing high–level concurrency in Java programs from version 5.0 on. Currently, Java SE 8 also offers parallel syntactical constructs for new multicore architectures programs, such as lambda functions with an efficient *tail recursion* implementation or execution to future.

Within a similar *historical* context, as in the Charles Dickens' *A Tale of Two Cities*, there may exist equivalent problems in two close–up places but they can be solved in a quite different way. In this respect, we are performing a systematic comparative study between two programming languages that are intensively deployed in industry today, probably due to their popular parallel programming APIs, Java Concurrency Utilities and C++11 Standard Library, respectively. We carry out a set of measurements to compare the execution time and performance (*speedup*) of parallel versions of (a) Belousov-Zhabotinsky chemical reaction simulation with cellular automata and (b) Discrete 2D-Convolution.

Differently from other similar work in the field, we focus here more on foretelling performance results that different APIs are offering to non-specialized programmers than in low–level implementation details.

Keywords: High-level concurrency API; multicore programming; parallel programming; simulation of physical phenomena; Belousov-Zhabotinsky's reaction; parallel 2D convolution

Introduction

For years, processor evolution has followed a direction towards increasing the number of processing units or *cores* as a way to break the power wall. Moreover, the growing presence of multicore processors pervades almost all areas of software development today. Multicore and GPU programming requires the integration of the ability to create and control threads ([10], [11], [19] and [25]) in several programming languages. This trend not only gives the programmer of multicores the ability to parallelize tasks on personal computers, but also to widespread parallel processing for carrying out high complexity calculations.

In a global *multicore and GPU programming scenario*, it may however become very difficult to know how to choose the appropriate programming language to get a sequential code parallelised well up to N processors. We believe that there is still little fundamental research done on how they compare different modern concurrent programming languages and concurrency APIs and on how they differently implement similar parallel and syntactical constructs. We perform measurements to compare performance and execution time of two programming languages, Java and C++, that offer popular parallel programming APIs, such as Java Concurrency Utilities and C++11, respectively.

Among the considered more competitive concurrent programming languages at present, to determine which one is offering better results regarding the shorter execution time and performance when writing parallel code for multicore architectures is still a pending issue.

There have been different solutions for the development of concurrent sotfware, offering diverse results to the application programer,

- Concurrency with processes using the *fork()* system call. This approach that can be considered obsolete at present unless used on purpose to carry out low–level programming for efficiency reasons. Lightweight processes (threads) might be not created at all, and we external libraries for synchronization or sharing memory between processes are needed. System V IPC specification and, more recently, POSIX IPC ([22]) are normally used to follow this approach, which presents integration issues with object oriented programming languages.
- Concurrency with threads by using the IEEE POSIX 1003.1c features. In this case, it can be used *phtread.h* library, thus providing the programmer with facilities for thread creation and management, synchronization control through basic locks and condition variables ([22]), which nevertheless may lead to *race conditions* between threads.
- Proprietary tools for parallel and distributed application development, including direct control of parallelism using an specific purpose sets of directives. MPI and OpenMP ([2]) are classic examples of this, and both of them have bindings for languages like C, C++ and Fortran.

From ISO/IEC 14882 C++ Standard Library [8] 2011 review (2014 amendment does not add any modifications to this), C++ programming language has now a set of capabilities within the new API for creation and control of concurrent threads that can be homologous to other general purpose languages ([25]) and concurrent object-oriented ones ([8] and [25]). 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 architecture programs, such as lambda functions with an efficient *tail recursion* implementation or execution to future.

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 the problems tackled here 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 2D convolution calculus with square matrices. We perform measurements that exclude overheads due to thread creation and launching, which could make different the deployment and concurrent management of thread models in both languages.

Differently from other similar work 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.

A comparative study of two problems will be developed in the following sections: (a) Belousov-Zhabotinsky chemical reaction simulation with cellular automata and (b) Discrete 2D-Convolution. Section II discusses the necessary tools to program concurrent/parallel tasks in both languages. Section III focuses on a parallel simulation of problem (a) with cellular automata. Section IV does the same as above for problem (b). Section V discusses the obtained results for the parallelizations carried

out in both languages, Java and C++. Finally, section VI brings out some conclusions regarding the observed execution times and performance in the two cases and sets the basis for future work.

Method

We have chosen the *execution time performance* measurement as the fundamental method to test and compare the concurency APIs of both languages, C++11 Standard Library and Java SE 8.0 Concurrency Utilities. The benchmarking programs have been the parallel Belousov-Zhabotinsky chemical reaction and Discrete 2D-Convolution programs, which will be discussed in sections and . *Speedup* or execution time performance is selected as the principal testing criterium since the most important outcome when developing for multicore processors is program execution speed. Other criteria such as power and memory resource management will also be of great interest to us. However, the availability of power and memory resource information is limited and very difficult to log. It must be part of future work since we need to change somehow the benchmarking method and task monitoring model to get a handle of this information.

Task Creation

The new C++ Standard Library 2011 defines the *Thread* class that allows the programmer to comfortably create a thread in the program code [8]. Two techniques are described in [25] to send a code segment that turns into a program's thread (see Annex 1.1),

- To pass the concurrent code to be executed to a constructor function as an argument, whose formal is a function pointer.
- To use a lambda–function that allows to wrap the code directly, i.e., without resorting to any reference–function that contains the code. This is often used when the code is passed internally ([25]).

Java also uses two techniques [14] [15] [19] to implement *thread* creation. The first one consists of extending the class `Thread` that is specific to this language and does not have a C++ equivalent. The second one amounts to the implemention of a `Runnable` interface, which is conceptually similar to the previous function pointer–based technique in C++. Java thread creation is more time–consuming that the C++11 one, as it requires some extra work done by the JVM and OS and thus yields latency into the application's requests processing. Our approach here is known *Task-Centric Parallelism*, because it aims at handling tasks instead of creating threads. Threads are *pre-created* and behave as workers which are already functioning as idle threads. In our benchmarking model, workers obtain a task to execute and when it is finished up another task or *idles* is reassigned, instead of stopping and then starting another thread for each new task.

Task Management

We briefly discuss here the main issues regarding concurrent task management in both languages, since they will impact our benchmarking method.

When it comes to developing parallel programming, but especially when we go "concurrent" and the various threads share information, we will always have to face the problem of concurrent access to data, which could result in inconsistent results if it is not properly tackled. This shortcoming is always due to the model of memory type used by the programming language. The Java execution engine tends to have a *volatile* copy of the variables that any task uses, thereby causing subsequent inconsistency w.r.t the most recent updates of the copies. Java offers the programmer the *modifier* `volatile`, which avoids local copies that are not up to date, and forces all the threads to read and write data on the main memory directly. In the end, this does not solve our problem, since the current

specification (and implementations thereof) of Java language does not establish the atomicity of read/write operations.

On the other part, in C++ language, the memory model behaves alike in terms of non–atomicity when it comes to common variables accesses, and thus it shows the same effects of overriding and inconsistency on shared data. In consequence, in both languages, Java and C++, the availability of concurrency control constructs becomes compulsory for data sharing and mutual exclusion access, such as critical regions that may contain variables atomically shared by threads.

Some other relevant aspects regarding state changes of concurrent tasks in both languages are summarized next,

- C++11 schedules an execution event to run a thread since the object thread is created, whereas Java requires to send an explicit signal to the Java virtual machine (JVM) through the `start()` method, then causing that a created thread could be dispatched for execution.
- Method `join()` gives the necessary control to the main thread for waiting completion of a set of daughter threads in both languages.
- Equally, both languages define methods to pause a thread for a given time, e.g.,`sleep ()` method, with a comparable semantics.
- Java provides the class–method `setDaemon()` to turn any thread into a (usually) long duration *daemon* thread, e.g., to perform supervision or maintenance tasks throughout the entire life–cycle of the application.
- C++ offers a similar functionality by the invocation of `detach ()` method.

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. We can see the real evolution of the BZ reaction on a *Petri* dish in Figure 1.

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

Figure 1 BZ Chemical Reaction within a Petri dish.

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. With these symbols, the quantities of A, B and C at time $t+1$ can be written as:

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

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 (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:

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

[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 the central cell), and divided through by 9.

With the above equations, we can derive the desired algorithmic simulation shown in Annex 1.2. The *compute* function is the most relevant part of the algorithm:

```
void compute (){
  for (int x = 0; x < width ; x++){
    for (int y = 0; y < height ; y++){
      float c_a = 0.0, 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 between the threads to move from one state[2] to the next one will be required.

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

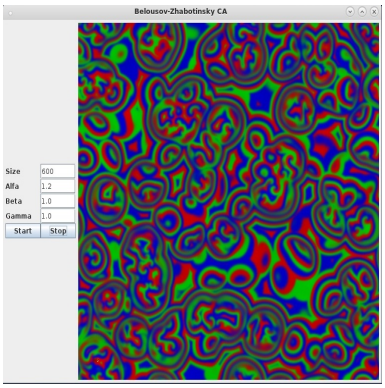

Figure 2 Simulation of BZ Chemical Reaction with Celullar Automata.

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.

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:

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

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 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 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. No condition of synchronization between the threads is necessary here. Parallel convolution only processes data array one time.

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.

Results

Parallel simulations of BZ and convolution were run on four nodes of our University cluster. Each cluster node has two Intel Xeon 2.6 GHz processor with Hyperthreading disabled, 128 GB RAM and 20 MB smart cache (L3), which amounts to 16 physical cores available in total. All nodes are running Red Hat Enterprise Linux 6.4 OS. The input nodes run Server with High–availability version and the *"compute nodes"* run Compute Node software. The management of each *compute node* is made by CMU (Cluster Management Utility) provided by Hewlett-Packard. For both problems solutions, initial data arrays were preloaded with random data and the following compiler versions,

- Oracle's SDK version 1.7.0_40.
- GNU's GCC 4.4.7.

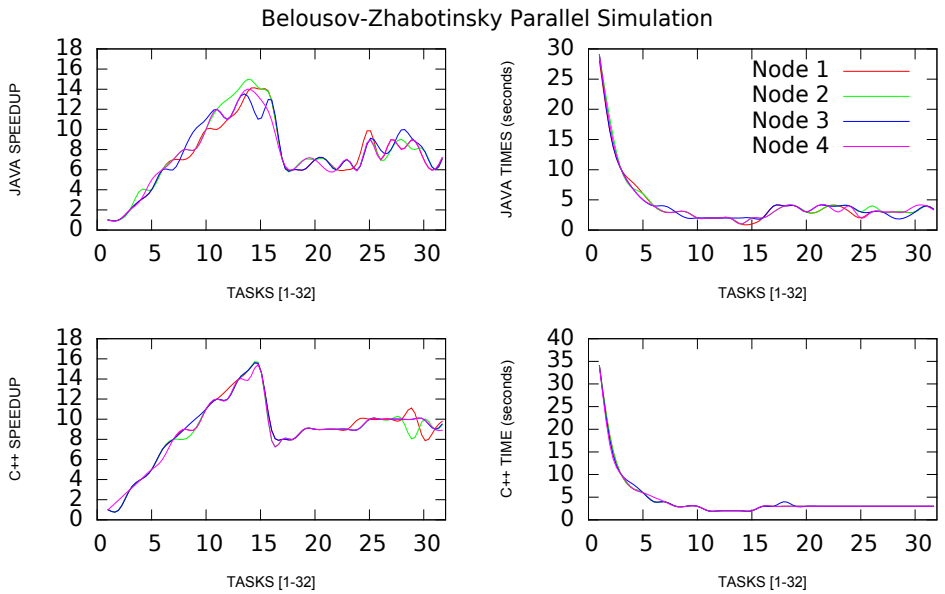

Figure 3 Execution time and sppeedup for BZ Parallel Simulation

Tasks	Java mean speedup	C++ mean speedup	Java mean time (s)	C++ mean time (s)
1	1	1	20,85	4,88
4	4,24	4,16	4,94	1,18
8	9,33	8,57	2,23	0,58
12	12,88	13,22	1,63	0,38
16	16,53	16,98	1,26	0,29
20	7,35	14,00	2,84	0,35
24	7,08	13,88	2,94	0,35
28	8,09	16,38	2,58	0,30
32	8,80	17,16	2,37	0,29
Mean relative std. deviation	3,45%	6,51%	3,09%	5,79%

Table 1 Java vs. C++11 in 2D Convolution (sample)

The simulation was run on four cluster–nodes with a 1600×1600 matrix each one. The reaction evolution was computed for 100 generations and no data were shared among nodes. The four nodes executes the full simulation. We show the data obtained for each of the nodes in Figure 3 which shows the speedup as a function of the number of tasks, for both Java and C$+ + 11$ languages.

An thus, we can see how the parallel simulation speedup raises as the number of tasks is increased. The parallel simulation program yields the best performance when only one task per core is allotted, as some prior similar work [23] [11] and [12] reported. From this point on, we see that increasing number of parallel tasks worsens the measured speed up, which finally results on a plateau. We can see that C++11 is providing better speedup results than Java, already showed result by other authors ([12]).

In some samples, after running the program many times, we have obtained hyperlinearity cases, i.e. values of speedup $\geq T(1)/T(n)$ (best execution time for one thread $T(1)$ and n threads $T(n)$), due to data locality when the cellular automaton fetches data from immediate neighbours.

Figure 3 shows the simulation execution time as a function of the number of tasks, for both Java and C++11 languages. The best parallel processing run time is also achieved when only a task is running per physical core. To scale the number of tasks beyond 16 does not improve the execution

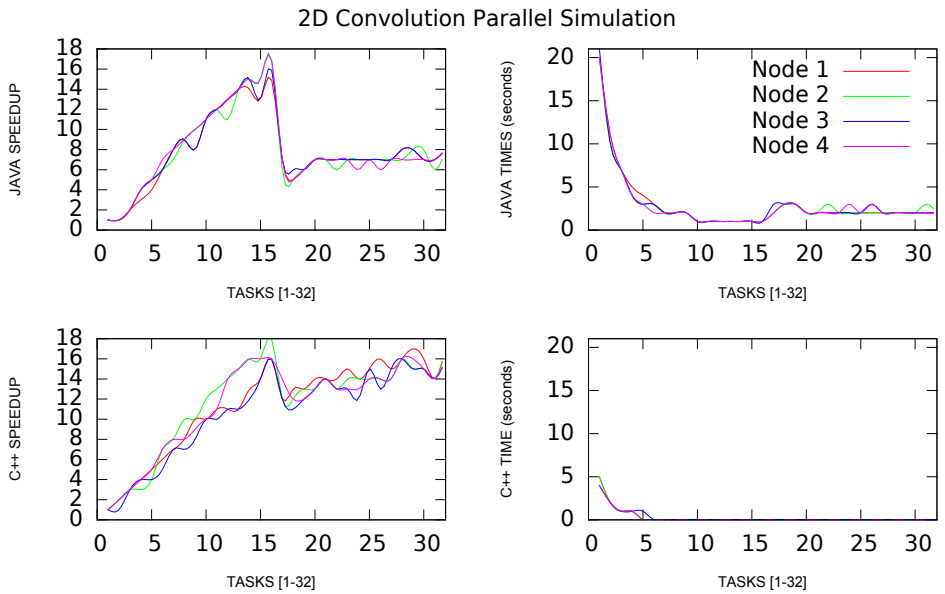

Figure 4 Execution times and speedup for 2D Convolution Parallel Simulation

Tasks	Java mean speedup	C++ mean speedup	Java mean time (s)	C++ mean time (s)
1	1	1	29,13	34,78
4	3,79	4,39	7,69	7,93
8	8,49	9,05	3,44	3,84
12	11,77	12,96	2,48	2,68
16	12,99	8,97	2,25	3,88
20	7,55	9,32	3,86	3,73
24	6,82	9,86	4,28	3,53
28	9,49	10,56	3,09	3,29
32	8,56	9,98	3,41	3,49
Mean relative std. deviation	3,45%	6,51%	3,09%	5,79%

Table 2 Java vs. C++11 in Parallel BZ (sample)

time results. Java's program usually gets equal or better computation times than the C++11 one, which may seem surprisingly odd since Java runs on top of the JVM and C++ compiles to native code. This behaviour can be explained if we take into account that threads processing in Java is performed by deploying a thread pool to manage the states of task life–cycle. This technology can achieve a significant improvement in the execution time [11] [23] when used in multithreaded programs. There is not exist a standard thread pool implementation in c++. Instead, to avoid the creation new threads for each generation, we have synchronized the threads using a barrier so that each time a thread computes all its assigned values it waits until the rest of the threads finish the computation of the rest of the generation.

As it can be seen in Figure 4, the obtained results for 2D Convolution are very similar to the showed by BZ simulation. In the convolution case, C++11 shows better results for speedup and execution time than Java does, especially when the number of threads exceeds the number of available cores per processor. Hyperlinearity phenomena were also observed in some run samples, and they occurred for a task number slightly higher than in BZ simulation executions. Recall that convolution only makes one processing pass on the data data arrays while BZ makes one hundred passes. Also, the

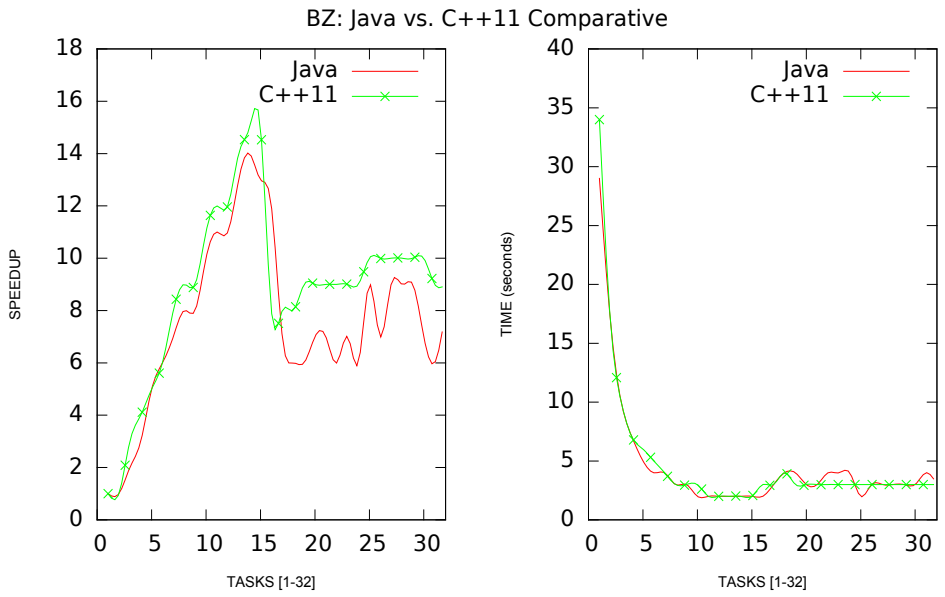

Figure 5 Java vs. C++11 in Parallel BZ

convolution parallel program requires two data arrays while BZ simulation will require three, which are three-dimensional arrays.

A comparative analysis of BZ parallel simulation, for both languages, is shown in Figure 5. In that figure, we can see that speedup results are slightly better when we use C++11 for a number of tasks smaller than the number of physical cores. Besides, the trend remains over the entire task number range where C++11 implementation performs better. An analysis of the execution times reveals an analogous behavior. There is a tiny advantage of Java over C++11, similiar to the showed in [12], if the number of tasks remains below 16, and, virtually, there exists an identical behaviour if the number of tasks raises above 16 tasks. As expected [11] [23]), to run the parallel program with a task number higher than the number of cores does dramatically worsen the behaviour of both variables, speedup and runtime. This is probably due to the increased overhead yielded by thread dispatching at low level.

Finally, we can see the comparative analysis for parallel 2D Convolution in Figure 6. If we take the speedup results from a qualitative point of view, we can conclude that the obtained performance is equal to the previous BZ–simulation case. However, from a quantitative point of view, we can notice several changes. We can see now that the gap between Java and C++11 is greater than in BZ case. Again, although a small gap can be seen near the best value, the difference is accentuated when the number of threads exceeds the available number of cores per processor. This result is not surprising at all, since for the BZ parallel simulation, one hundred of generations were carried out on the data arrays, and the life cycle of parallel tasks was managed by a thread pool, which is not currently available in C++11 Standard Library. When enabled, the Java thread pool is more efficient regarding thread management than the corresponding C++11 implementation, since threads in the Java pool can be reused, and thus some advantage is obtained over C++11 threads measured performance. Regarding the execution times, and differently from BZ parallel simulation, Java behaves worst than C++11 throughout the entire task number range. Analogous to the prior discussed case, the reason owes to the thread pool deployment in the Java implementation. BZ–simulation is performing one

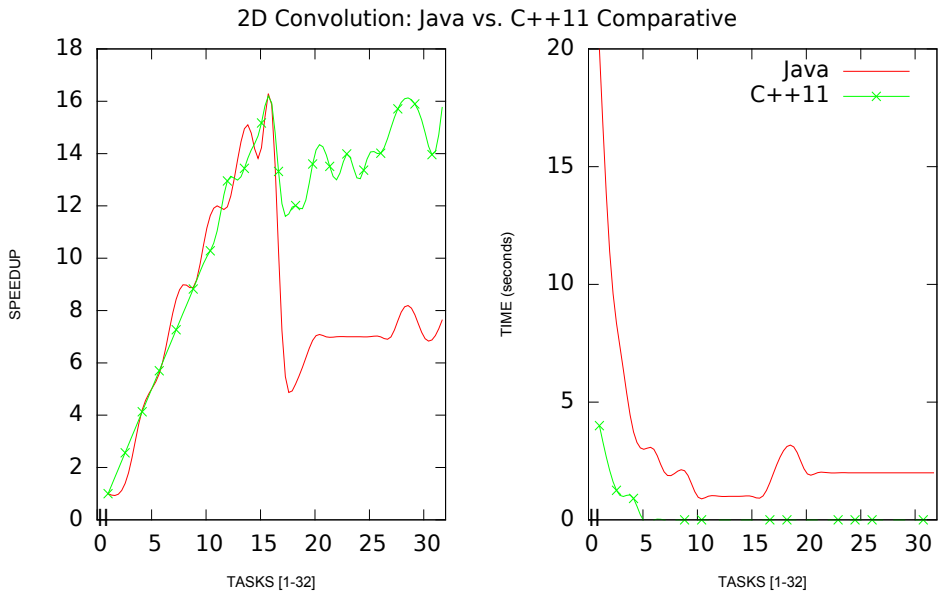

Figure 6 Java vs. C++11 in Parallel 2D Convolution

hundred iterations on data and, consequently, the synchronization overhead is much higher in the BZ–simulation than in the 2D convolution. In such case, the pool helps to improve the execution time obtained. With 2D-convolution implementations, to create a thread pool is not worthwhile, because it is very demanding in terms of runtime and then Java must pay a non rewarded price that finally affects program performance.

Conclusions and Future Work

Parallel implementations of BZ simulation and 2D–Convolution are proposed. The first implementation is made by approaching the theoretical model of the reaction with a cellular automaton. The second one, 2D Convolution implementation, is a straightforward and standard program. Parallel implementations in Java and C++11 are written for both problems above, then a comparative analysis of speedup and execution time is carried out.

The obtained results showed that for problems with relatively short processing time solutions, C++11 always behaves better than Java with respect to the two variables analysed here. However, calculations for BZ–simulation showed that for more dense data clouds, requiring higher processing times, the Java programming language yields better execution times than C++11 does, though the obtained speedup results worst.

We can therefore conclude that in problems needing massive numerical computation to be solved, Java is at least as efficient as C++11 in terms of execution time. This fact is likely to continue midterm, at least until C++ incorporates the *thread pool* construct to manage the life–cycle of parallel tasks, which has been scheduled for 2017.

Our future work will tackle aspects regarding how power and memory resources could affect parallel multicore and GPU programming in both programming languages.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Department of Computer Science, University of Cádiz,Avda. de la Universidad de Cádiz, 10, 11519 Puerto Real, Cáadiz, Spain.
[2]Department of Computer Science, University of Granada, Granada, Spain.

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 Phemomenon to 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 super-linear 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 Investigation of the Tools of Seismic Data Processing. Proceedings of the 8^{ht} Biennial International Conference & Exposition on Petroleum Geophysics, 303-309, Hyderabad, 2010.

1 Annex

1.1 Tasks creation in c++

Using the thread constructor:

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

Lambda function:

```
#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 : myThreads){thread.join();}
  return(0);
```

```
}
```

1.2 BZ simulation

```
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++){
      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;}
 }
}
```