**ANN. MULT. GPU PROG.**

# A Parallel Model for the Belousov-Zhabotinsky Oscillating Reaction with Python and Java

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. Cádiz, Spain

[2]Department of Software Engineering University of Granada. Granada, Spain

**Abstract**

The programing language Python has been rapidly gaining in popularity and it has now become the first choice for implementing all kinds of systems in different software development fields. Programmers now use it for parallel processing on multicore and manycore architectures through specific modules such as Numba, PyCuda or mpi4Py. Much analysis work has been conducted to compare the performance of Python and commonly-used programming languages such as Java. This article presents a further comparison by solving the Belousov-Zhabotinsky oscillating reaction problem with both languages by using symmetrical multiprocessing with data partition.

**Keywords:** Belousov-Zhabotinsky oscillating reaction, barrier, concurrency, threads, Java, multiprocessing, Python, processes, speedup locks, mutual exclusion, synchronization

## 1. Introduction

Python is an extremely powerful, dynamic programming language which is quickly becoming widely used in all areas of technical and scientific computing, including parallel processing on all kinds of platforms in order to improve application performance [2]. It was ranked first in The 2017 Top Programming Languages Classification [19] followed by C [5] and Java in second and third place. As a multipurpose language, Python provides native support to parallel processing through two modules: threading and multiprocessing. The first of these enables multithread programming which is very similar both in terms of use and philosophy to other languages such as Java, C # and C ++ [9, 15, 16], and incorporates all the standard support for synchronizing threads with locks, traffic lights, and condition variables. Nevertheless, it is useless to attempt efficient parallel programming with Python threads [23] since there is an intrinsic restriction that prevents it. The Python interpreter incorporates a global interpreter lock (GIL) to prevent parallel thread execution. In fact, the GIL forces a thread to acquire a lock for execution, and stops concurrent access to Python objects from different threads, thereby protecting the interpreter memory and causing the memory collector to work in a suitable way. The diagram in Figure 1 shows how the Python GIL works for three parallel threads.

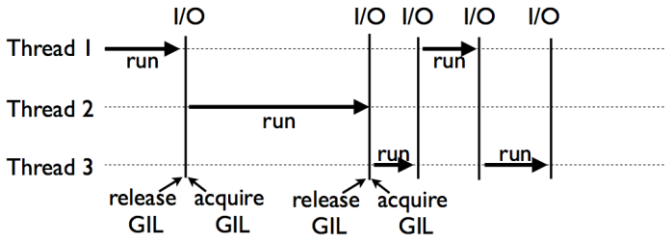


**Figure 1.** Python thread execution with GIL

There is a very clear conclusion to be drawn from the thread time line: it is not possible to develop parallel programming with threads using Python. If we attempt to do so, we will obtain sub-unit speedups.

The alternative is for programmers to use a parallel process approach that does not require the use of GIL. This article presents a simulation model for the Belousov-Zhabotinsky chemical reaction using a bi-dimensional, cellular automaton. For comparison purposes, we implement the model with Python and the popular Java in order to compare both parallel implementations. Our results show that Python offers worse execution times and speedups than Java, something which is consistent with other kinds of problems as Table 1 shows.

The article is structured as follows: Section 2 briefly describes the Belousov-Zhabotinsky chemical reaction; Section 3 provides a basic background to cellular automata; Section 4 describes the methodology used; Section 5 presents and discusses our measurements; and finally, Section 6 outlines our conclusions.

## 2. Introduction

The Belousov-Zhabotinsky reaction (BZ) was the first oscillating chemical reaction to be discovered by P.B. Belousov in 1950 when he dissolved citric acid in water with acid bromate and ceric ions to produce an oscillating pattern of yellow tones (Figure 2) as the reaction time progressed. Nowadays, there are many industrial chemical processes with oscillating reactions and some of these have a great financial impact. For this reason, it is extremely interesting to describe this type of reaction with mathematical models and to quickly and efficiently solve these models.

If two substrates A and B are evenly distributed in a reactive medium, they produce a new substrate C according to the equation $A + B \rightarrow C$.

**Figure 2.** BZ reaction in a Petri dish

When the reaction occurs in a Petri dish, a characteristic oscillating pattern with complex waves is obtained (Figure 2). It is very simple to write the concentration time variations for the three substrates using the following dynamic system approach, where the reaction speed $r$ depends on concentrations A and B:

$$\dot{A} = -r$$
$$\dot{B} = -r$$
$$\dot{C} = r$$

From this, it is easy to see that $r = kAB$ where the $k$ constant is usually determined experimentally. All of these concepts can be extended to include the presence of stoichiometric coefficients according to the equation $\alpha A + \beta B \rightarrow \gamma C$, where α, β, γ≥0. The law of mass action allows us to reformulate $r$ as $r = kA^{\alpha}B^{\beta}$ and so we can logically write

$$\dot{A} = -\alpha r$$
$$\dot{B} = -\beta r$$
$$\dot{C} = \gamma r$$

and trivially

$$\dot{A} = -\alpha k A^{\alpha} B^{\beta}$$
$$\dot{B} = -\beta k A^{\alpha} B^{\beta}$$
$$\dot{C} = \gamma k A^{\alpha} B^{\beta}$$

for initial concentrations of substrates $A_0$, $B_0$ and $C_0$. These equations can eventually be solved and expressed in the following discrete way:

$$A_{t+1} = A_t + A_t(\alpha B_t - \gamma C_t)$$
$$B_{t+1} = B_t + B_t(\beta C_t - \alpha A_t)$$
$$C_{t+1} = C_t + C_t(\gamma A_t - \beta B_t)$$

In these, the amount of each solute at time $t + 1$ depends on the amount of the solute at time $t$, and the concentrations of the other two solutes adjusted by stoichiometric coefficients.

## 3. 2d cellular automaton

There are many definitions of cellular automata in literature. In many fields, they have been used as a tool to model highly complex physical realities such as the spreading of forest fires, substance percolation, combining solutes from a chemical reaction or simulating urban traffic. We have chosen the definition established in [30] and applied it to simulate BZ oscillating reactions. A cellular automaton (CA) will be defined as a 4-tuple $M = (\zeta, \varepsilon, N^{I}, \rho)$ where:

- ζ is a regular discrete network of cells (also called nodes) together with a set of border conditions for the finite case which are used to define the neighborhood of cells located at the network border.
- ε is a finite set (usually with an algebraic Abelian ring structure) of states that the network cell can adopt.
- $N^{I}$ is a finite set of cells that define the neighborhood with which a given network cell can interact.
- ρ is the transition function, that defines how a cell's state can change according to time and the state of its neighboring cells $N^{I}$.

In view of these definitions, any cell area can be defined as a network t included in the real $R^{d}$ and which uniformly covers a portion of the *d*-dimensional Euclidean space. Each cell is labeled by its position $r \in \zeta$. The layout of the cells is spatially specified by their connections with their closest neighbors and these connections are obtained by connecting pairs of cells following a regular pattern. For any spatial coordinate $r$, the neighborhood grid $N_b(r)$ consists of a list of neighboring cells defined by

$$N_b(r) = \{r + c_i : c_i \in N_b, i = 1, \cdots, b\}$$

where $b$ is the coordination number (i.e. the number of the nearest-neighbors in the grid that directly interact with the cell located at coordinate $r$). We use $N_b$ to denote any nearest-neighbor pattern with the elements $c_i \in R^d$ for $i = 1, \cdots, b$. For our model, we choose *d=2*, so

$$\zeta = \{r: r = (r_1, r_2) \in Z^2\}$$

The total number of cells available is usually denoted by $|\zeta|$. In computer simulations, AC use finite grids ($|\zeta| < \infty$), and border conditions must be imposed. Our simulation will use *Moore's* border conditions. The set of neighboring cells whose state influences a given one is defined by the interaction neighborhood $N_b^I(r)$ for a given $r$ cell, according to the following expression:

$$N_b^I(r) = \{r + c_i : c_i \in N_b^I\}$$

There are a number of ways to choose this neighborhood and for our simulation, we have chosen the Moore neighborhood where any cell only has its surrounding cells as neighbors (Figure 3).

| | | |
|---|---|---|
| $X_{i-1,j-1}$ | $X_{i-1,j}$ | $X_{i-1,j+1}$ |
| $X_{i,j-1}$ | $X_{i,j}$ | $X_{i,j+1}$ |
| $X_{i+1,j-1}$ | $X_{i+1,j}$ | $X_{i+1,j+1}$ |

**Figure 3.** Moore neighborhood for cell $r = (i, j)$

Each cell $r \in \zeta$ has a state $s(r) \in \varepsilon$. The elements in set ε can be numbers, letters, or symbols. An overall configuration of the automaton $s \in \varepsilon^{|\zeta|}$ is determined by the state of all the cells in the grid. Finally, the time-evolution dynamics of our model is determined by the function of transition ρ that specifies the changes in any cell state according to its previous state, and the interaction with its nearest-neighboring cells and this is given by:

$$\rho: \varepsilon^{\mu} \rightarrow \varepsilon$$

where $\mu = |N_b^I|$. The rule is proved to be spatially homogeneous and does not therefore explicitly depend on the position of a given cell. Extensions of the definition to include temporary or spatial homogeneity are feasible. If the CA is deterministic, the function of transition yields only one feasible change of state, whereas if it is stochastic, the new state of a cell state is given by a specific probability distribution.

## 4. Method

From the discrete-equation model of the BZ reaction and using a two-dimensional cellular automaton [4, 11, 12, 19, 21], it is possible to write an algorithm that is capable of simulating the time evolution of the reaction. Its pseudocode is illustrated below.

```
Algorithm BZ

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;}
 }
}
```

Using this code, we developed and subsequently parallelized programs written in Python (Figure 4) and single-thread Java to solve the simulation through symmetric multiprocessing with automatic data partitioning [3, 15] according to the number of tasks. With Python, and given the very serious limitations imposed by the global interpreter lock for parallel threads to be efficient, we chose to use the multiprocessing module to model tasks with processes, and the NumPy module so that the standard arrays were efficient. In Java, tasks were supported with the Runnable interface. For both languages, the tasks were executed using a thread pool executor [7, 8, 10, 13, 20]. It was also necessary to introduce a barrier

synchronization condition to resynchronize the tasks after each stage of the simulation.

## 5. Measurements and discussion

In order to develop the measurements, we used an Intel (R) Core ™ i5-5440 processor (3.10 GHz) with 8 GB of RAM and the Linux Fedora 24 operating system. We used Python version 3.6.4 and Java version 1.8.0_111 from Oracle. The grid size was 1600×1600 nodes, and the simulation was made for a total of 100 iterations. Time measurements were taken for a variable number of tasks between 1 and 16, and these were used to calculate the resulting speedups. The measurement results are displayed in Figures 5 and 6.

The first characteristic that we should highlight is the very wide difference in execution times between both languages. This might well attract attention if it were not for the fact that it has been well described in literature [17] for a very wide range of problems. Table 1 lists some of these problems and shows the times required to solve them using the two languages and the amount of memory required.

| ***Problem*** | ***P(s)*** | ***J(s)*** | ***P(Mb)*** | ***J(Mb)*** |
|---|---|---|---|---|
| PIDIGITS | 3.43 | 3.12 | 12.7 | 36.7 |
| REGEX | 15.22 | 10.34 | 447.3 | 627.2 |
| REV.-COMP. | 3.26 | 1.03 | 264.5 | 191.0 |
| K-NUCLEOTIDE | 77.65 | 8.70 | 182.7 | 375.5 |
| BINARY-TREE | 93.55 | 8.34 | 280.6 | 293.0 |
| FASTA | 59.47 | 2.33 | 15.9 | 43.9 |
| MANDELBROT | 225.24 | 6.04 | 15.7 | 76.5 |
| N-CUERPOS | 838.39 | 22.10 | 10.3 | 33.1 |

**Table 1.** Python versus Java

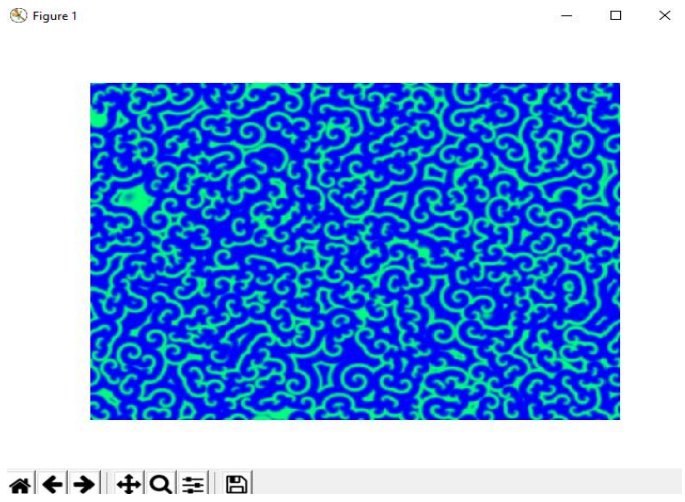


**Figure 4.** BZ simulation with Python sequential code

As the figure shows [17], Python is always slower than Java although it generally uses less memory to solve the problems. It is also apparent how as the computational load of the problems to be solved increases, this difference becomes even greater and is particularly pronounced when generating the Mandelbrot set or in the numerical resolution of the *n*-bodies problem.

Figure 5 shows the execution times for an increasing number of tasks and we can see how differences between both languages compare favorably with those already illustrated in literature for very different types of problems from those in Table 1 while fully consistent with it. It should also be noted that Python behaves particularly well

when the number of parallel processes is equal to the number of physical cores available on the platform, which in this case is four, while it proves to be very sensitive to a larger number of tasks, with worse times for these situations.

In view of this, is the limitation presented by Python's symmetric multiprocessing so serious? We believe that the answer to this question is yes, but with some important differences since Python provides scientific computing-oriented modules which can greatly improve processing times. As an example, in addition to the NumPy module that we have used, the SciPy module enables even greater improvements. Table 2 displays the times obtained by rewriting the sequential version of Python using SciPy.

| ***Sequential Version*** | ***Time (seconds)*** |
|---|---|
| Python | 282.39 |
| Python (SciPy) | 29.56 |
| Java | 7.15 |

**Table 2.** Python (with/without SciPy) versus Java

As can be seen, by introducing the SciPy module and using the predefined 2-d convolution included to recalculate solute concentrations, it is possible to accelerate the sequential version of Python to a tenth of the original time brining it close to Java.

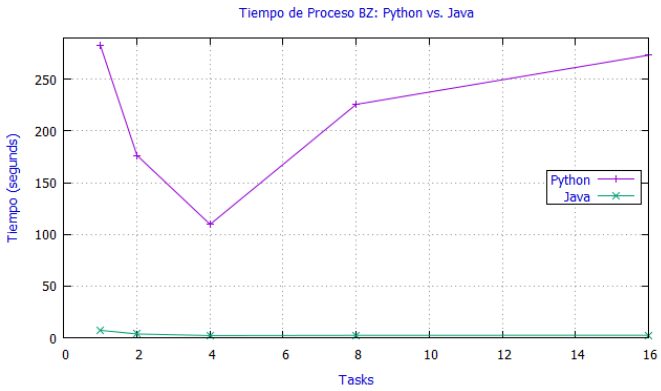


**Figure 5.** Processing time for BZ parallel simulation

Figure 6 illustrates the speedups obtained for the processing times for a different number of tasks when compared to a sequential version of the simulation. We can see that Java accelerates computations better than Python, and that the plateau stage that is reached for this type of curve when tasks exceed the number of physical cores in Python slopes slightly where acceleration decreases. In both languages, optimal speedup is clearly achieved when the number of tasks equals the number of physical cores in line with similar types of problems to the one we are dealing with.

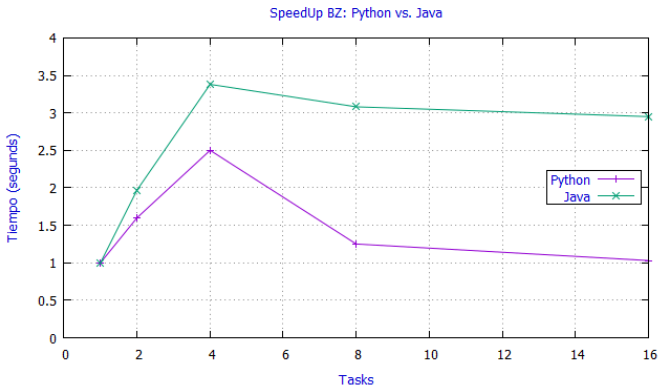


**Figure 6.** Speedups for BZ parallel simulation

## 6. Conclusions and future works

From the results obtained, we can conclude that the symmetric multiprocessing with processes in Python does not improve either the time or acceleration of the results obtained with Java for the BZ problem and this is fully consistent with literature for a wide range of different problems as Table 1 shows. Nevertheless, it is still possible to further accelerate the code in Python by using specially designed, scientific computing modules which can reduce the execution time to up to a tenth to obtain execution times for parallel solutions that approach and match those obtained with other languages used in symmetric multiprocessing.

Our future work will focus on designing simulations of the BZ reaction on GPU architectures using the Python language and the Numba and/or PyCuda modules.

## References

1. Adamatzky, A., De Lacy, B. & Asai, T. Reaction-Diffusion Computers. Elsevier B. V., 2005.
2. Akhter, S. & Roberts, J. Multicore Programming Increasing Performance through Software Multithreading. Intel Press, Digital Edition, 2006.
3. Balaji, P. (ed.) Programming Models for Parallel Computing. The MIT Press, 2015.
4. Bandman, O. Mapping Physical Phenomenon to CA-Models. Automata-2008. Theory and Applications of Cellular Automata, 381-395. Luniver Press, 2008.
5. Batty, M., Memarian, K., Owens, S., Summit, S. & Sewell. P. Clarifying and Compiling C/C++ Concurrency: from C++11 to Power. POPL'12. (http://www.cl.cam.ac.uk/~pes20/cppppc/popl079-batty.pdf)
6. Boehm, H. Threads Basics. What Every Programmer Should Know About Memory Models Issues. HPL Technical Report, 2009 (http://www.hpl.hp.com/techreports/2009/HPL-2009-259html.html)
7. Fernández, J. Java 7 Concurrency Cookbook. Packt Publishing, 2012.
8. Göetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. y Lea, D. Java Concurrency in Practice. Addison-Wesley, 2006.
9. 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.
10. Lea, D. Programación Concurrente en Java. Principios y Patrones de Diseño. Addison Wesley, 2000.
11. Mikhailov, A. S. and Showalter, K., Control of waves, patterns and turbulence in chemical systems. Physics Reports 425, 79-194, 2006.
12. 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.
13. Oaks, S. & Wong, H. Java Threads, 3rd Edition. O'Reilly, 2004.
14. Rauber, T. & Rünger, G. Parallel Programming for Multicore and Cluster Systems. Second Edition. Springer-Verlag, 2012.

15. Ringler, Roger. C# Multithreaded and Parallel Programming. Packt Publishing, 2014.
16. Robbins, K. & Robbins, S. Practical Unix Programming. A Guide to Concurrency, Communication and Multithreading. Prentice Hall, 1996.
17. The Computer Language Benchmarks Game, 2017. In https://benchmarksgame.alioth.debian.org/.
18. The Top Programing Languages 2017. In https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017
19. 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.
20. Subramanian, V. Programming Concurrency on the JVM: Mastering Synchronization, STM and Actors. The Pragmatic Programmers, 2011
21. Yoshida, R. Self-Oscillating Gels Driven by the Belousov-Zhabotinsky Reaction as Novel Smart Materials. Advanced Materials, Vol. 22, issue 31, pp. 3463-3483, 2010
22. Zaccone, G. Python Parallel Programming Cookbook. Packt Publishing, 2015