TECHNICAL UNIVERSITY OF CRETE

Electrical and Computer Engineering

Forecasting Promising Biological Simulations at PhysiBoSS

Diploma Thesis

Effrosyni Anesti

Advisory Committee: Antonios Deligiannakis, Associate Professor (Supervisor)
Michail Lagoudakis, Associate Professor
Vasilios Samoladas, Associate Professor

Chania, February 2020

To Vaggelis

Abstract

Since the existing biological multicellular systems are characterized by high complexity and heterogeneity, coupled with the fact that there has been a remarkable upsurge in computer science, in-silico methods based on mathematical models are in a great use. Specifically, they are particularly helpful when we must deal with diseases that have abnormal and unpredicted behavior, such as cancer or auto-immune ones. The need for understanding and curing such kind of diseases, led us to the integration of different modelling frameworks that take into account the intra- and the extra-cellular environment, as well as, the interplay between cells. Such an example is PhysiBoSS, that combines two other well-established frameworks to support its whole functionality and provide us a cell-fate decision model with an accurate representation of cells' population variance through the time under a specific treatment and conditions.

Considering the fact that not all PhysiBoSS simulations are hopeful, to facilitate the procedure of results' collection and examination, the bad simulations must be excluded. This thesis' goal is to design a distributed and parallel system that implements a forecasting algorithm on a great amount of real-time running simulations and decides about the sustainability or not of a simulation's execution and finally maintains only the top k hopeful ones out of all the initial simulations. The algorithm's performance was evaluated both locally and remotely/distributively, giving us very positive results.

Keywords

PhysiBoSS, parameters, TNF, pulse, cancer, cells, alive, simulation, forecasting, clustering, data analysis, time series, distributed system

Περίληψη

Το γεγονός ότι τα υπάρχοντα βιολογικά πολυκύτταρα συστήματα χαρακτηρίζονται από υψηλή πολυπλοκότητα και ετερογένεια, σε συνδυασμό με τη σημαντική εξέλιξη της επιστήμης των υπολογιστών, οδήγησαν στην αυξημένη χρήση των in-silico μεθόδων, βασισμένων σε μαθηματικά μοντέλα. Συγκεκριμένα, είναι ιδιαίτερα χρήσιμες όταν πρόκειται να μιλήσουμε για ασθένειες με μη φυσιολογική και απρόβλεπτη απόκριση, όπως είναι ο καρκίνος ή τα αυτό-άνοσα νοσήματα. Η ανάγκη για την κατανόηση και θεραπεία τέτοιου είδους ασθενειών, οδήγησε στη δημιουργία διαφορετικών εργαλείων μοντελοποίησης, τα οποία συνυπολογίζουν το ενδο- και εξω-κυτταρικό περιβάλλον, καθώς και την αλληλεπίδραση μεταξύ των κυττάρων. Ένα τέτοιο παράδειγμα είναι το PhysiBoSS, το οποίο συνδυάζει δύο άλλα ήδη σαφώς ορισμένα εργαλεία, για να υποστηρίξει την όλη λειτουργικότητά του και να παράξει τελικά, ένα μοντέλο απόφασης περί κυτταρικής μοίρας μέσω μίας ακριβούς αναπαράστασης της μεταβολής του πληθυσμού των κυττάρων στο πέρασμα του χρόνου, υπό ορισμένες συνθήκες και θεραπεία.

Λαμβάνοντας υπόψιν το γεγονός ότι δεν είναι ελπιδοφόρες όλες οι προσομοιώσεις του εργαλείου, προκειμένου να διευκολύνουμε τη διαδικασία της διαλογής και μελέτης των αποτελεσμάτων, οι κακές προσομοιώσεις πρέπει να εξαιρεθούν. Επομένως, ο στόχος της παρούσας διπλωματικής εργασίας είναι η σχεδίαση ενός παράλληλου και κατανεμημένου συστήματος, το οποίο εφαρμόζει έναν αλγόριθμο πρόβλεψης σε ένα μεγάλο πλήθος τρεχουσών προσομοιώσεων και αποφασίζει για τη συνέχιση ή όχι της εκτέλεσής της κάθε μίας και τέλος ανιχνεύει και κρατά μόνο τις k πιο ελπιδοφόρες εκ του ομαδοποιημένου συνόλου προσομοιώσεων. Η απόδοση του αλγορίθμου ελέγχθηκε τοπικά και απομακρυσμένα/κατανεμημένα, αποδίδοντας θετικά αποτελέσματα.

Λέξεις Κλειδιά

PhysiBoSS, παράμετροι, TNF, παλμός, καρκίνος, κύτταρα, ζωντανά, προσομοίωση, πρόβλεψη, ομαδοποίηση, ανάλυση δεδομένων, χρονοσειρές, κατανεμημένο σύστημα

Acknowledgements

Firstly, I would like to keenly thank my supervisor, professor Antonios Deligiannakis, both for his supervision and his valuable advice given on my subsequent career. I also want to thank him for his consistency, as well as, for being present anytime I needed him, during the preparation of my thesis work.

To continue with, I would like to specifically thank Mr. Mpotsis Ioannis, member of the Special Technical Laboratory Staff, since his help on issues considering the Grid Computer of the Technical University of Crete was highly important.

Moreover, I would like to thank Mrs. Arapi Xenia, member of the Laboratory Teaching Staff, for her help on various practical issues that emerged during the making of my thesis work and the postgraduate student and friend of mine Kontaxakis Antonios, who gave me great help and sped up the process of completing my thesis, by transmitting to me useful for my thesis knowledge, as well as, for his mental encouragement and support.

In addition, I owe a great thank to my professors, in general, and the other members of this department, who contributed to my diploma acquisition and my forming as an engineer.

Last but not least, I want to thank my family for all the mental and material support during the whole process of my studies and especially, my father, whose contribution was immense, my fellow student Rastsinskaya Karina, who helped me finish successfully and in time my studies, for our collaboration in most of the group assignments and the laboratory tasks of our department, as well as, Anastasia Palieraki, an important figure for me, who supported me and helped me in many ways.

Contents

List of Figures

List of Tables

1. Introduction

More and more efforts and attempts are being made from the scientific community to utilize *in-silico* instead of the so far being used in-vivo and in-vitro methods. The in-silico methods are taking place in virtual biological systems and experiments are performed on computer or via computer simulations. Such an example is *PhysiBoSS* which is a C++ software for multiscale simulation of heterogeneous multi-cellular systems.

PhysiBoSS is the combination of MaBoSS, a C++ software for simulating continuous or discrete time Markov processes, applied on a Boolean network, and PhysiCell, a physics-based cell simulator. Briefly, PhysiBoSS simulates cancer cells and their biochemical environment, as well. Thus, by running thousands of different simulations, initialized with different conditions, we are able to generate thousands of different results and examine them, deciding which conditions are promising and which are not promising, regarding the cancer cells' behavior. The effectiveness of this implementation has been checked and verified with a cell-fate decisions model in response to TNF (Tumour Necrosis Factor) injection. The problem that arises here is the fact that, by running a large number of different simulations, finally we get a high volume of data which cannot be examined one-by-one.

This work focuses on solving that problem by proposing a system for detecting the non-promising simulations and killing them. To be more specific, we designed a distributed system in which a number of different modules read data in parallel. Each one of them maintains the top k most promising simulations, killing on the fly the rest of the running simulations, that do not seem to belong to the top k ones.

To verify the functionality and efficiency of our algorithm, we tested it on approximately 80 GB (GigaBytes) of data, locally and remotely, as well, and the results seem promising enough, since approximately the 80% – 90% out of all the starting simulations get killed before the simulation reaches the 40% of the whole procedure, accelerating it.

1.1 Motivation

Due to the rapid, unpredicted and abnormal development of cancer cells, the need for representation, integration and treatment of cancer with software tools, was born. There are several modelling frameworks that have already been integrated and are in use. One of them is PhysiBoSS that can approach and depict the complexity and heterogeneity of multicellular biological systems at a quite satisfying level. What this simulator, finally, offers is a detailed description about each cell's response over time under some well-defined conditions. Its results are very realistic, although there are significantly numerous different combinations that can be made and, by extension, numerous outputs that are generated. So, the problem that emerges here is that the results are too many to be examined. For this reason, we propose an algorithm for forecasting biological simulations within this tool and which can, of course, be used and expanded to other use cases, as well, to offer a more clear view about which the proper conditions for cancer cells to get eliminated are. Thus, the examination of the results would be much easier to be conducted, gathering only the promising simulations. So, the fact that we

could pitch, at least a little bit, in the effort related to cancer research and treatment, would be an honor for us and here is where the whole motivation lays.

1.2 Outline

In the current thesis work, in *Chapter II* all the background theoretical knowledge and the software tools that were used are mentioned and explained. Furthermore, in *Chapter III* an extensive explanation about our whole's system architecture, as well as, our algorithm's implementation is provided. After that, all the experimental, including quantitative and qualitative results, are illustrated in *Chapter IV* and we sum up with the conclusion and future work in *Chapter V*.

2. Background

We need to mention and clarify some very important for our design tools that were used and combined to achieve our goal, so as the readers of this thesis be able to keep pace with our conceptualization.

2.1 Biological Tools

More and more scientists focus on the creation and development of biological tools that are used in medical research so as to virtually represent through simulations the very complex and multicellular biological systems (in-silico) on their attempt to find solution on diseases like cancer and eliminate as well, the in-vivo applied methods. The significance of such tools can be conceptualized if we only think that we didn't know that cells existed until microscopes were invented. Respectively, living in the epoch of big data and informatics, the design of software biological tools is the logical flow of things.

2.1.1 MaBoSS/MaBoSS 2.0

MaBoSS [2] stands for Markovian Boolean Stochastic Simulator and as its name evinces, it is a C++ software open source that simulates continuous or discrete time Markov processes, which are applied on a Boolean network, namely, it is an environment for stochastic Boolean modeling. It applies Monte-Carlo algorithm so as to produce time trajectories, according to some initial conditions [7]. The motivation is to create a model of signaling pathways, namely, a model that represents the behavior of cells and provides probabilities about model entities (gene, protein etc.) reaching a stable state.

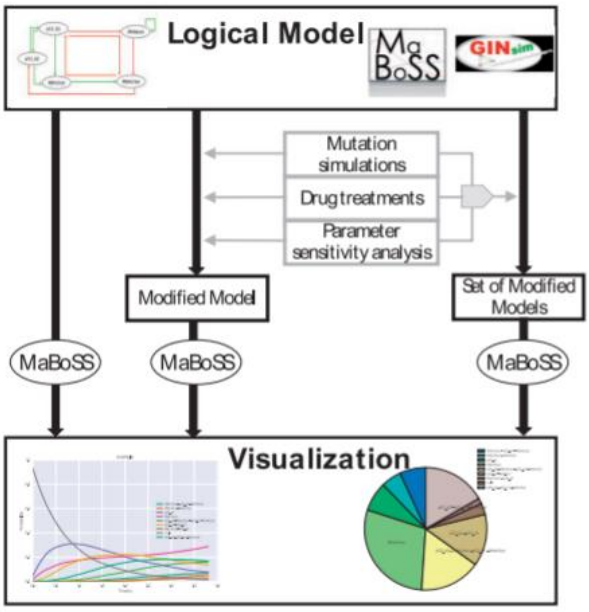

Figure 2.1: Pipeline of the use of MaBoSS 2.0 [2]

The figure above [Figure 2.1] showcases a MaBoSS logical model which is initially modified by some factors (mutations, drug treatments, sensitivity analysis) and after some logical equations, finally they gathered in a single command and simulated all together at once.

MaBoSS has already been considered as a well-defined tool and it is widely used in biological problems like the cancer-related ones. MaBoSS 2.0 is just an extended version of MaBoSS. Some of the new characteristics that embeds are visualization, mutations' simulations, drug treatments etc.

2.1.2 BioFVM

BioFVM [4] is a parallelized diffusive transport solver for 3D biological simulations. To be more specific, it is a well-defined and efficient simulator for varying diffusing substrates by cell and bulk sources in large 3D domains. A 1-hour simulation's example is illustrated at Figure 2.2.

BioFVM is a minimal-dependencies simulator, written in C++ and parallelized with OpenMP. It can be executed on its own or can be adopted and included in other larger modelling packages, too.

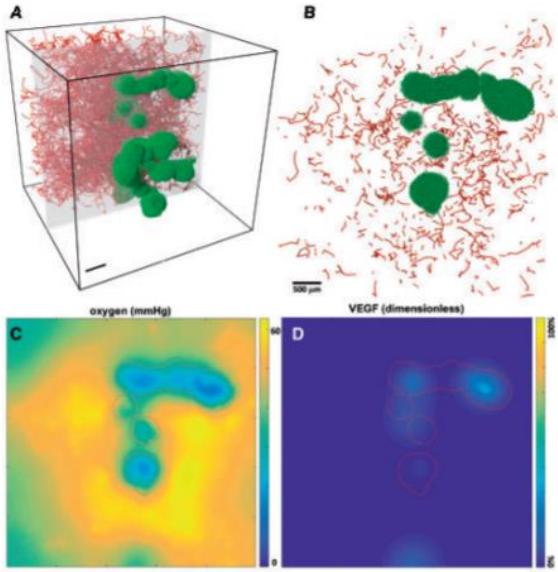

Figure 2.2: Simulation of oxygen and VEGF diffusion in a large scale [4]

2.1.3 PhysiCell

PhysiCell [3] is an open source framework for 3D multicellular simulations. It is a multi-agent-based modelling tool that embeds the before-mentioned tool, BioFVM and simulates the tissue-scale behaviors that arise from biological and biophysical cell processes representing each individual cell/agent as a physical dynamical entity. A simulation's example is depicted at Figure 2.3. PhysiCell is an agent-based software with minimal dependencies written in C++ and is parallelized with OpenMP [8].

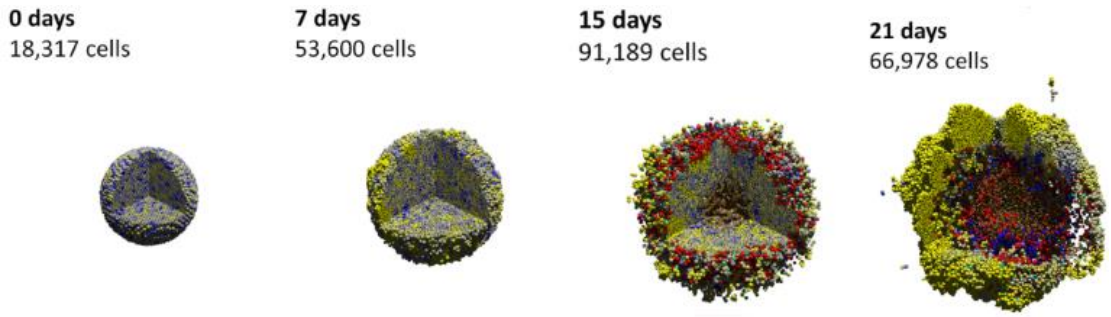

Figure 2.3: 3D simulation of adaptive immune response to a heterogeneous tumor Simulation of oxygen and VEGF diffusion in a large scale [8]

2.1.4 PhysiBoSS

The motivation behind the creation of PhysiBoSS is the fact that the multicellular biological systems are very difficult to be represented due to their complexity and heterogeneity. Since these systems involve high inter-dependency among the different biological scales of a cell to be developed, only a multi-scale mathematical model would be appropriate for that kind of representations. So, its representation with a well-defined tool that takes into account cell signaling, cell population response and extracellular environment is highly significant for the scientific community which occupies with the comprehension, integration and discovery of cancer.

PhysiBoSS [1] (depicted at Figure 2.4) is a C++ software framework that was developed in 2019. It is an agent-based modelling tool that is used for multiscale simulations of heterogeneous multi-cellular systems. To be more specific, it creates a virtual representation of a multi-cellular system, namely, the cells, their intra- and extra-cellular environment, as well as, the interplay between cells and their surroundings. It is an agent-based mathematical model and each agent represents an independent cell.

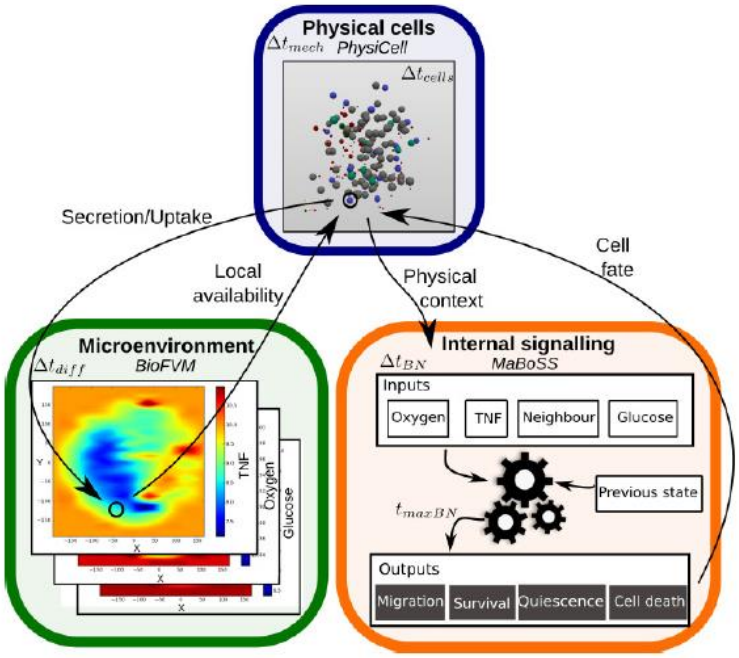

Figure 2.4: Schematic Representation of PhysiBoSS [1]

The results, after running a PhysiBoSS simulation is to obtain a detailed description of each agent/cell's response through time. The whole implementation of PhysiBoSS is represented with a cell-fate decisions model in response to Tumour Necrosis Factor (TNF), a multifunctional cytokine that plays an important role in diverse cellular events.

To achieve such a kind of functionality PhysiBoSS is based on two (2) other already integrated and well-defined software frameworks that we previously presented; MaBoSS and PhysiCell and more specifically is an adapted version of PhysiCell in which inside each agent/cell, Boolean network computation is integrated. PhysiBoSS is the tool, the simulations' results of which provided us the data, that we handled.

Running a simulation

Before we continue, we need to explain how a PhysiBoSS simulation starts running so that the readers can familiarize themselves with the following work. PhysiBoSS is an open source tool and can be found on GitHub [9].

PhysiBoSS consists of three (3) executable files:

- *PhysiBoSS*, which handles the real simulation,
- *PhysiBoSS_CreateInitTxtFile*, which generates an initial cells' condition of a simulation,
- *PhysiBoSS_Plot*, which depicts the cells' position and condition of a simulation at specific timepoint.

To properly run a simulation, we need to follow some steps. First of all, simulations are distributed into well-organized folders regarding the changeable parameter in which we will refer soon. Inside the main folder is included another folder named *BN* which contains the Boolean network configuration files. In addition to that, the main folder contains the *runs* folders, as well. Inside each run folder an individual simulation is included. Each simulation must be prepared before we start running it. Concretely, inside this folder, we need to generate the initial conditions of cells. That can be achieved either with the *PhysiBoSS_CreateInitTxtFile* executable or by using an already generated one text file, named *init.txt*. Furthermore, the most important file that each simulation must contain is the *parameter.xml* file [10], which is the file in which all possible parameters are defined, namely, the properties of this specific simulation. The parameter file is composed of four (4) different parts, according to the type of the parameters to define; simulation, cell properties, network and initial configuration. The *simulation* element contains the simulation's global properties (e.g. output_interval, maximal_time). The *cell properties* element is related with properties which are common to all cells (e.g. protein_threshold, cell_radius). In the *network* section of the parameter file, the properties relative to the Boolean network computation (e.g. network_update_step) are defined and in the *initial configuration* parameters like tnf_concenctration, as well as, the init.txt file we produced. Finally, each run's folder must contain an empty *output* and microutput folder in which the simulation's output files will be stored. An example about the precise organization of the simulations is demonstrated at Figure 2.5, while Figure 2.6 depicts how a parameter file looks like. More details about parameters, can be found in reference.

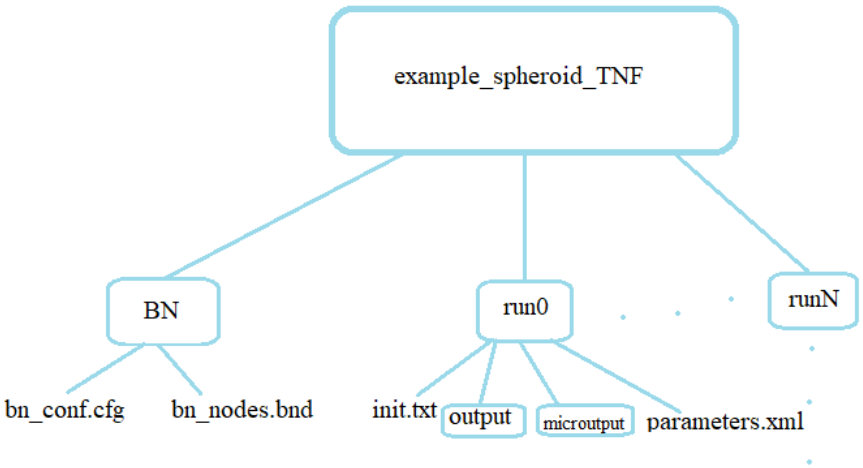

Figure 2.5: Folder organization of an example

```
<?xml version="1.0" ?>
<!-- Global parameters of the simulation (time parameters are in min, distance in microns) -->
<simulation>
        <!-- diffusion time scale -->
        <time_step> 0.02 </time_step>
        <!-- Time scale of motion, cell volume changes -->
        <mechanics_time_step> 0.1 </mechanics_time_step>
        <!-- Cell cycle time scale, change of cell phase -->
        <cell_cycle_time_step> 2 </cell_cycle_time_step>
        <!-- Time to simulate, 24 h -->
        <maximal_time> 1440 </maximal_time>
        <!-- Write output file of cells positions every 30 min -->
        <output_interval> 40 </output_interval>
        <!-- parallelization with openmp -->
        <number_of_threads> 2 </number_of_threads>
        <!-- parallelization with openmp -->
        <mode_cell_cycle> 1 </mode_cell_cycle>
        <!-- To use with a boolean network implementation -->
        <output_densities> 450 </output_densities>
        <!-- parallelization with openmp -->
        <write_passive_cells> 0 </write_passive_cells>
        <!-- Discretization size of the microenvironment grid (BioFVM) -->
        <minimum_voxel_size> 15 </minimum_voxel_size>
        <!-- Dimensions of the simulated space -->
        <bounding_box_xmin> -500 </bounding_box_xmin>
        <bounding_box_xmax> 500 </bounding_box_xmax>
        <bounding_box_ymin> -500 </bounding_box_ymin>
        <bounding_box_ymax> 500 </bounding_box_ymax>
        <bounding_box_zmin> -500 </bounding_box_zmin>
        <bounding_box_zmax> 500 </bounding_box_zmax>
        <!-- Densities to simulate in the microenvironment -->
        <number_of_densities> 2 </number_of_densities>
        <density_0> oxygen </density_0>
        <density_1> tnf </density_1>
        <!-- Constant injection of densities on the external boundaries -->
        <dirichlet_boundary> 0 </dirichlet_boundary>
        <!-- When output densities files, don't write all the voxels values, but randomly selected ones -->
        <write_ratio_voxels> 0.4 </write_ratio_voxels>
</simulation>

<!-- Properties of the first cell line, common to all cells in this type -->
<cell_properties>
        <!-- How much polarized by default -->
        <polarity_coefficient> 0.1 </polarity_coefficient>
        <!-- Motility parameters -->
        <motility_amplitude_min> 0.01 </motility_amplitude_min>
        <motility_amplitude_max> 0.01 </motility_amplitude_max>
        <mode_motility> 1 </mode_motility>
        <!-- Cell-cell adhesion and repulsion coefficients -->
        <homotypic_adhesion_min> 2 </homotypic_adhesion_min>
        <homotypic_adhesion_max> 2 </homotypic_adhesion_max>
        <heterotypic_adhesion_min> 2 </heterotypic_adhesion_min>
        <heterotypic_adhesion_max> 2 </heterotypic_adhesion_max>
        <max_interaction_factor> 1.2 </max_interaction_factor>
```

Figure 2.6: Sample of a Parameter File

2.2 Distributed Messaging System & Libraries

For the needs of this thesis, we used a distributed messaging system, as well as, some libraries, so as to achieve the desirable results.

When we are dealing with large amount of data, we have two (2) main challenges. The first challenge is how to collect a large volume of data and the second challenge is to analyze the collected data. To overcome those challenges, we need a messaging system. A messaging system is responsible for transferring data from one system/application to another allowing them not to worry about how to reliably share data, namely, how to send, distribute, store or manage data. So, a messaging system is useful when we deal with processing messages. When we are talking about a distributed messaging system, then we mean that the messaging system is a robust queue, with the FIFO (First-In First-Out) architecture, containing messages. Messages are queued asynchronously between client applications and messaging system. A distributed messaging system provides the assets of reliability, scalability and persistence. Some of the most popular messaging systems are Kafka, RabbitMQ, JMS (Java Message Service).

There are two (2) types of messaging patterns; Point-to-point and publish-subscribe messaging system. In both patterns, producers send messages and consumers receive them.

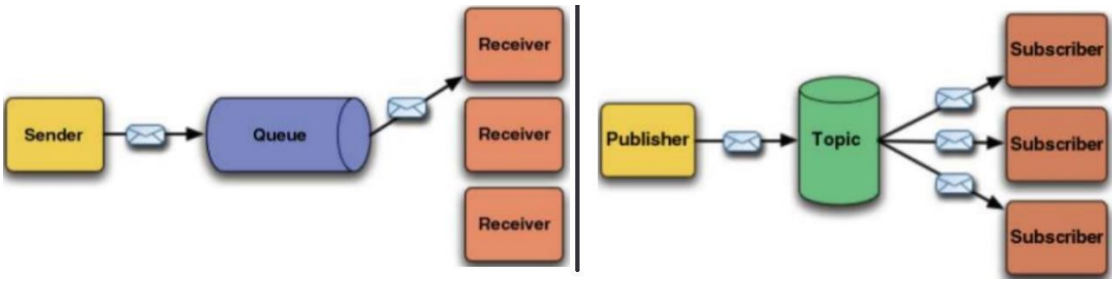

Figure 2.7: Point-to-point (left) and pub-sub (right) architecture [11]

On the one hand, in *point-to-point* (left part of Figure 2.7) communication, producers called senders produce and send a message to a queue and exact one (1) consumer called receiver can consume/receive this specific message.

On the other hand, in publish-subscribe (pub-sub) communication, depicted at the right part of Figure 2.7, all the messages are sent from producers - here called publishers - to a topic and consumer - here called subscriber - can subscribe to one or more topic and consume all the messages in this topic. As a result, a message can be consumed from more than one consumer asynchronously and consumers are able to consume messages simultaneously or not.

The most commonly used architecture is the pub-sub one, since it gathers a lot of benefits, some of which are robustness and scalability, it improves testability and facilitates asynchronous workflows, as well as it supports both offline and online service.

2.2.1 APACHE Kafka

An example among others of a distributed messaging system follows the pub-sub model is Apache Kafka. Apache Kafka is a distributed publish-subscribe messaging system and a robust queue that can handle a high volume of data and enables us to pass messages from one endpoint to another. Kafka messages are persisted on the disk and replicated within the cluster to prevent data loss. It is mainly designed for distributed high throughput systems. Furthermore, we must mention that Kafka is built on top of a third-party tool, ZooKeeper server, to achieve load balancing. Briefly, ZooKeeper is a centralized synchronization service which allows a configuration maintenance within distributed systems (multiple clients to perform simultaneous reads/writes/acts, keeps track of Kafka cluster etc.).

Figure 2.8 shows the Kafka architecture.

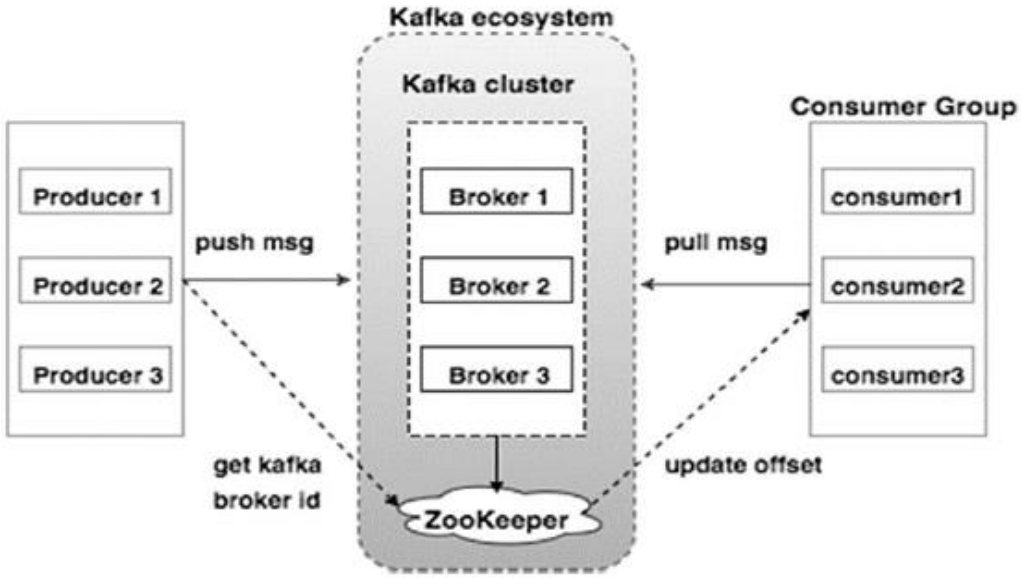

Figure 2.8: Kafka Ecosystem [15]

Briefly, Kafka consists of certain parts and works as following. One or more Producer-Publisher instances produce and push messages/records to Kafka. Kafka is a distributed system, so it can run in multiple different nodes consisted a cluster. Kafka's cluster stores the received messages in different categories called topics and separates one from another by name. Each topic is divided into several partitions, depicted in Figure 2.9, and each one of them can be regarded as a FIFO queue and distributed across brokers who represent the cluster's nodes and are responsible for maintaining the published data. Thus, splitting data into multiple brokers (nodes) we achieve to parallelize a specific topic consuming messages included in it from different machines simultaneously or at different times. Consumers pull messages of the topic they subscribe and start reading data at any time point they want. Finally, it is not necessary for publishers and consumers to run at the same time. Partitions can be replicated if the user wants to. To be more specific, partitions are distributed across brokers, so each broker holds a number of partitions which can be either the *leader* partition of the topic, which means this is the real single partition in which a producer writes or the replica partition which is an exact copy of the leader partition (Figure 2.10). In this way, we

keep backup of each partition and as a result of all data sent from producers, preventing data loss as well as from leader potential failing (fault-tolerant).

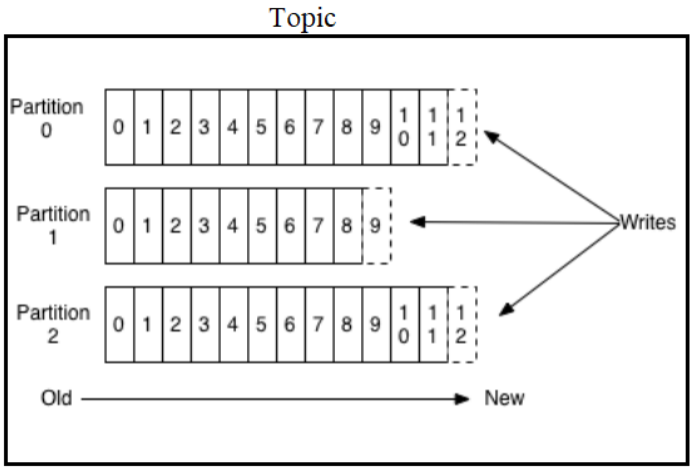

Figure 2.9: Anatomy of a Topic [12]

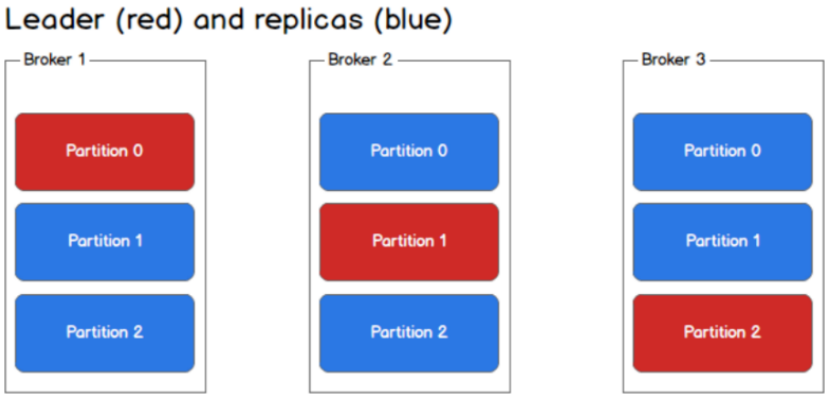

Figure 2.10: Partitions & Brokers [12]

To sum up, the reason why we chose Kafka for our design is because apart from gathering all the benefits of a distributed messaging pub-sub system, it is also very fast since it performs 2 million writes per second and it provides high throughput and low delay rendering it suitable for real-time data processing.

2.2.2 Cppkafka/rdkafka

Regarding the fact that the data source we used is written in C++, for the producer's design we used the *cppkafka* library, which is freely available on github [16]. Cppkafka is a version of Apache Kafka, that allows C++ applications to produce and consume messages using Apache Kafka protocol. Cppkafka's design is based on the rdkafka library [17]. According to its creator, librdkafka is a C library implementation of the Apache Kafka protocol with high performance. Thus, cppkafka library consists a librdkafka's wrapper and provides high level consumer and producer interface, as well as, some other utilities like a buffered producer (which simplifies producer error handling) and a compacted topic consumer. Cppkafka lets us create Kafka producers/consumers and configuration with very little code.

2.2.3 Confluent Kafka

For the consumer's design we used Confluent Python Kafka [18] so as to make good use of some specific modules that python has at its disposal, ideal for time series data analysis (pandas DataFrame, Numpy, scipy). Confluent Kafka is one out of the three released and reliable python libraries high level clients for Apache Kafka; Kafka-Python, PyKafka and Confluent Python Kafka. The reason why we chose the last one, is because its performance is better than the other two. It is offered by Confluent Platform, a platform for real-time and historical events, to build real-time data pipelines and streaming applications, founded by the Apache Kafka's creators, as a wrapper around librdkafka that we mentioned before [page 18].

2.2.4 Pandas

The most important tool we used to append, manage and recall stored data was pandas [27]. In 2008, an open source software library written for the Python language, called pandas, was developed. To be more specific, pandas is a package for easy data manipulating and analysis, suitable for data science through Python. It provides high-performance, as well as, flexible and expressive data structures. It seems to be the fundamental high-level building block for doing data analysis using Python. The DataFrame [28] is one of the most important data structure pandas provides us and the main data structure we used during our design for the data analysis and processing part. It is a two-dimensional, size-mutable table in which data is stored in well-defined labeled axes (columns and rows). The structure can be accessed anytime either by using the axes' labels, the index or the specific numerical position.

2.2.5 Minidom

Minidom [29] stands for Minimal DOM (Document Object Model). It is a library provided by Python language, to describe, access, manipulate and process XML documents using its functions, properties and variables [30]. In case users do not feel comfortable with minidom, they can use the ElementTree of type Element.

2.3 Databases

A database (db) is an organized structure which includes stored data. The main structure to represent a database is a table, namely, an infrastructure which is modeled in rows and columns, to make processing and data querying efficient. Thus, we can store data and easily access, manage, modify, update, and generally, control them. There are two (2) kinds of database models; SQL (Structure Query Language) and NoSQL databases. For our architecture, we used a NoSQL database.

2.3.1 MongoDB

The last tool we used to complete our design was a NoSQL database, MongoDB [32], which is the most popular and commonly used NoSQL database. MongoDB is a document-based database. To be more specific, in MongoDB we can store data in JSON-like format and have the right to access them as well as edit them whenever and from wherever we need to.

Since we wanted to access MongoDB from a python script, we used PyMongo, a driver to access MongoDB from Python. The only limitation about its use, is that we have to first start a mongod instance, a server that runs as a background process, so as to handle data requests, manage data access and generally perform background management operations.

2.4 Forecasting & Time Series Analysis

2.4.1 Forecasting

Forecasting is the process of making predictions about the future response of the variable of interest by analyzing its past trends. Nowadays, forecasting is ubiquitous, since it is used and it is desirable to be used in many fields (e.g. finance, meteorology). There are many and different forecasting algorithms, some of which will be presented afterwards. A good general pattern we could use so as to be able to do a prediction is firstly, to visualize data in a time series plot and after that, capture and model the underlying patterns if these exist (e.g. sines). We could say that the two (2) prominent forecasting methodologies that exist, divided into two (2) types of classes:

1. **Classical/Statistical** (Moving Average, ARMA, ARIMA, Exponential Smoothing, Theta)
2. **Machine and Deep Learning** (Quantile Regression Forecast, Recurrent Neural Networks, seq2seq)

The question that arises here, is which one of the forecasting techniques and models is the best for the forecasting the use case that we are challenged to confront with. This depends on multitude of different factors, like how much historic data is available, which are our constraints, if there is any correlation with variables and others. So, to find the one that suits the problem we are dealing with, we have to compare the "nominated" different forecasting approaches. To achieve that we need to do chronological testing, named backtesting. To do that, we must train on a time series of events, up to a certain point and then tests subsequently. There are two (2) approaches of chronological testing, the sliding and the expanding window, when it comes for high frequency data testing and for data that does not come so often and the historical data points are limited, respectively. However, we will not do a further analysis on that issue at the moment. We just provide a figure for each method at Figure 2.11.

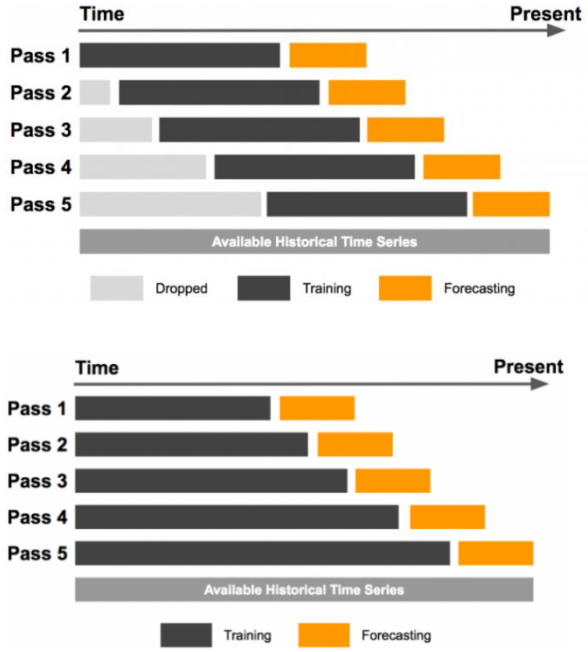

Figure 2.11: Sliding (up) and Expanding (down) Window [34]

2.4.2 Time Series Analysis

We already mentioned the significance of forecasting. We now expand to time series forecasting. A time series is a set of observations taken at specified times, usually, at equal intervals (e.g. x-axis gets the time, divided into equal intervals). Time series is used to predict the future values of a variable based on the previous observed values and generally the independent variable is time. So, time series analysis is the procedure in which we analyze time series data in order to come to some meaningful conclusions. Its significance lays on business forecasting, understanding past behavior, planning future, evaluating current accomplishments.

When we are dealing with time series, there are some characteristics that we need to consider. These components of time series are trend, seasonality, irregularity, cyclic, autocorrelation and stationarity. Briefly:

- Trend: the movement to relatively lower (downtrend) or higher (uptrend) values over a «long» period of time.
- Seasonality: a repeating pattern within a fixed time period (periodic).
- Irregularity: erratic response through time (noise).
- Cyclic: repeating up and down movements.
- Autocorrelation: similarity between some observations through time.
- Stationarity: when time series has a particular behavior over time, there is a very high probability that it will follow the same in the future, namely, the non-changing over time statistical properties of time series.

Although time series analysis is a very useful procedure, there are some cases in which we should not apply it, for example, when the values are constant or they are in the form of functions (e.g. sine).

Before we move on, we must inevitably take a moment to examine more elaborately stationarity which is a very important characteristic of time series analysis and must always be present. Stationarity has three (3) very strict criteria:

1. Constant mean/average according to the time
2. Constant variance, namely, the distance from the mean according to the time
3. Autocovariance which is independent on time

To check if stationarity is or is not present in our time series, we can check it either with rolling statistics or with DF (Dickey-Fuller) test. Briefly, the first method plots the moving average or variance, depicting if it varies through time. At the second one, we do the null hypothesis that our model is not stationary and after doing the test we get some critical results which depending on their values, evince if the model is stationary or not.

So, in this way we can check for our time series modelling stationarity presence and if it is not, then after some transformations we can make it stationary, so as to continue the procedure.

2.4.3 Time Series Forecasting Algorithms

As we already mentioned, there are lots of time series forecasting algorithms, statistical or machine or deep learning ones. Here we will introduce some already implemented and effective ones excluding the naïve classical forecast which is that today's value will hold for tomorrow.

Fast Fourier Transformation

FFT is an easy, understandable and simple way for forecasting. Actually, it is a decomposition into a series of sine functions, and we use it when periodicity exists. To be more specific, what we do is to run FFT on input data, filter out the low amplitude or the high frequency components, which are, apparently, the noise, we get some sinusoid functions from which we pick the first (1^{st}) *n* most significant ones, we apply on them forecasting (move our phase forward), we run the inverse of FFT on filtered data and finally we have the profit. Considering we are living in an imperfect world, the existed periodicity in the input data, will have deviation and as a result the forecasting data will have inexpediency. The solution to this problem is to iteratively compensate input data with error until no spikes exist.

Sequence to sequence (seq2seq)

It is a very powerful technique of forecasting with deep learning and it was published by Google to solve the machine translation problems. The key idea is the fact that by nature all the time series data we collect is not continuous. It is discretized data points on which we applied interpolation. So, what we do is to perform forecasting per input, since we have well-defined data points, by combining the history. To be more specific, the data points are dependent to each other. The value of the current time may depend on the value of the past timepoints. We can extend the past however long we want to. The context for each item is the output from the previous step. Thus, a neural network consisted of two (2) components, encoder and decoder, is drawn, depicted at Figure 2.12.

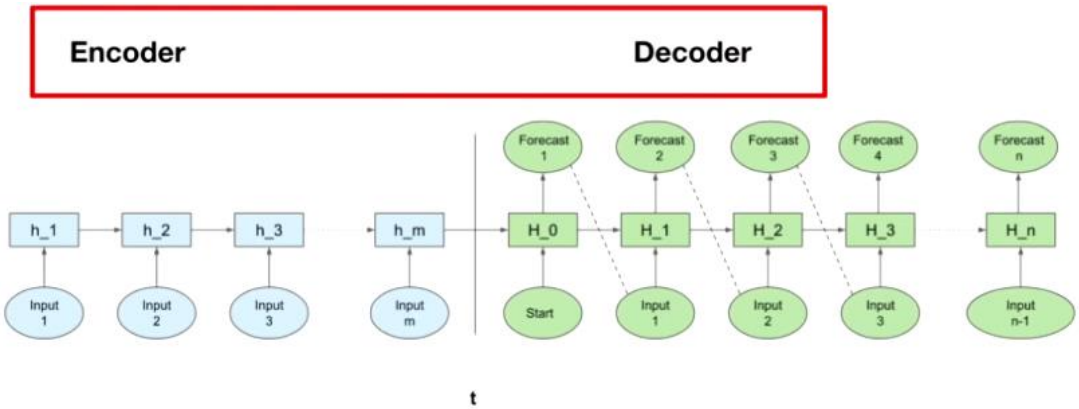

Figure 2.12: seq2seq Schematic Representation [35]

Moving Average

Moving average is one of the most known techniques in time series modelling. This approach just states that the next observations will be the mean of all the past ones. Although it is a very simple and naive approach, it is in use and it efficiently works.

ARIMA

ARIMA model stands for AutoRegressive Integrated Moving Average and it is an extension of ARMA (AutoRegressive Moving Average) model. In this kind of model, we assume that the current value depends on its previous values with its own lags. Because the term AR, ARIMA is a linear regression model. Afterwards, we add the moving average model we presented above and finally the order of integration to achieve the desirable prediction on the input data.

3. System Design

Reaching the implementation part, this thesis' goal was the development of a forecasting algorithm, which supports local and remote communication, and which keeps only the top *k* out of the *N* initial running simulations killing in real-time the rest ones.

3.1 Preprocessing

The very first thing we had to do was to find a way to easily generate thousands of different "parameters.xml" files (already explained in page 14). To achieve that, we used Python's module "minidom", which is a minimal implementation of the Document Object Model interface [29]. Briefly, we created a python script that, with the use of the minidom methods [30], describes the general structure of this xml file including all the different elements and initialized in the most frequently used values. After that, we created a second python script which according to the parameter or parameters that the user wants to change, as well as, the varying range and step each one of them, it calls the previous script and generates the different parameters.xml files. In addition to that, not only does it create them, but, also, it creates the whole tree of a completed example, containing all the necessary directories and files, as well as, allocating them properly into the run folders. After generating an adequate amount of ready-to-run simulations, we distributed them into a 44-node cluster, and received approximately 80 GB data with different parameters' combination. With the use of a python script we translated the data into figures so as to depict the simulations' response through time and to be more specific, the number of alive, apoptotic and necrotic cells within a day (1440 minutes).

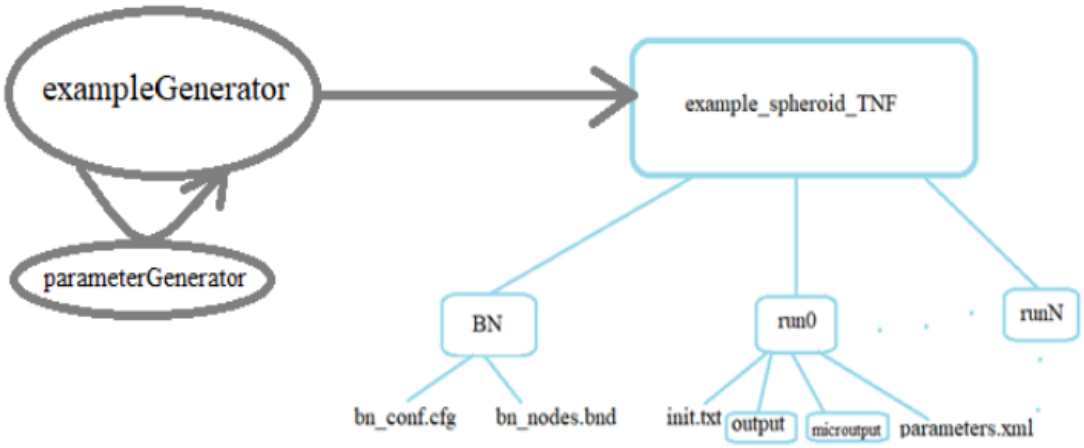

Figure 3.1: Example Generator

3.2 System Overview

Since having and knowing the efficiency of a high volume of data, we then proceeded to our system's design.

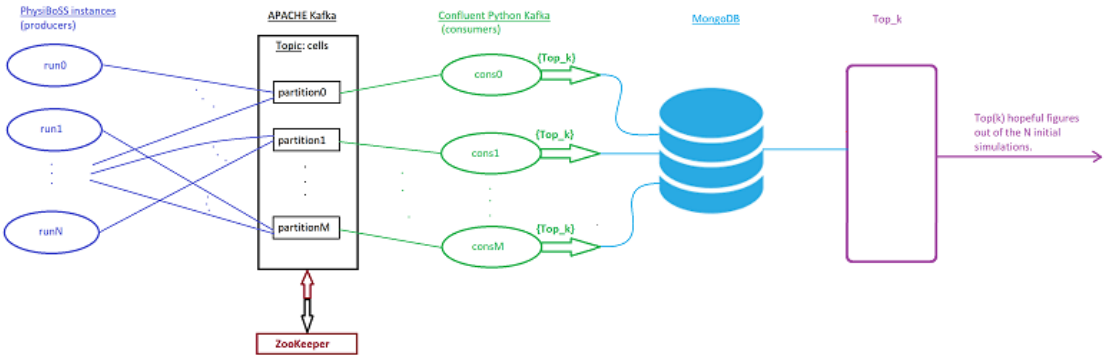

Figure 3.2: Whole Ecosystem

The algorithm should be used in a remote communication, too. So, we needed to create a distributed message passing system to support that. To achieve that, we used Apache Kafka (page 17). In particular, the whole system's design follows the structure depicted in Figure 3.2.

The concept is the following. We start running *N* different simulations. We distribute them in different machines to be executed. We must be able to handle them in a parallel way. Each simulation is manipulated by a specific consumer's instance. Each consumer's instance handles *X* different simulations and applies the forecasting algorithm on them. Locally, the consumers announce the top_*k* simulations and send them to a database. Finally, the last module TOP_*K* retrieves the locally top_*k* lists from the database and decides the final top_*k* out of the *N* initial simulations.

Hereupon, we discuss the individual parts.

3.3 Data Source

Firstly, we have the PhysiBoSS simulator and we need to send data to Kafka, by the time they are generated. We thus interloped inside the PhysiBoSS software and more specifically, within the *WriteCellReport* function which is implemented in the *cell_container.cpp* file, to create the producer, using a high-level producer architecture supported from cppkafka Apache Kafka client library, specifically designed for C/C++ applications. The producer's part includes the configuration part in which we define the broker or the brokers, namely, the machine's IP address in which ZooKeeper and Kafka servers are running, as well as, we set the port on which the Kafka server is listening for connections, allowing the broker to be moved to a different host/port without confusing consumers. Regarding this configuration, we create a producer instance. Also, if the desirable topic in which we want to send the messages does not exist, then we create it. After that we are ready to produce and flush the message to a specific partition. The partition number, according to our design, is randomly generated. So, we do not know in which topic's partition each simulation is. Although the producer's

configuration is ready to push messages to Kafka, before we send them, we need to edit them somehow.

3.4 Data Editing

PhysiBoSS generates data which are stored inside the two (2) output folders; microutput and output. The one that needs our attention is the output folder. In this folder are stored the data related to cells, while the other one gathers the molecules' information. The file that supports the information about cells is a .txt file and it looks like the one depicted in Figure 3.3. The rate that these files are generated is defined by the output_interval parameter the value of which is assigned inside the parameter.xml file. All the simulations are considered to last 1440 minutes (1 day). So, considering that the time step, meaning, the output_interval has the value 30 (minutes) then 49 files will be generated $(1440/30 + 1)$, one for each timestamp with the first one (1) come at timepoint 0 and the last one at timepoint 1440.

By the time a cells_*.txt file is generated, we read it line-by-line. From each line we extract only the useful for us information which is the *phase*. Phase carries the information about the state in which each agent/cell is oriented, for the specific timepoint we examine. The phase is represented with an integer code, each one of which describes a different state. We can see the correspondence among PhysiBoSS integer codes and PhysiCell cell states at Figure 3.4.

```
Time;ID;x;y;z;radius;volume_total;radius_nuclear;contact_ECM;freezer;polarized_fraction;motility;cell_line;Cell_cell;phase;Cycle;NFkB
0;0;-51.3206;-4.61239;-83.4215;8.5;2572.44;5.11093;0;0;0.1;0.01;0;2.6464;1;0;-1
0;1;-37.4339;-33.3428;-84.6856;9.8791;4038.69;5.94016;0;0;0.1;0.01;0;2.67582;0;0;-1
0;2;-37.66;-13.829;-86.4875;8.5;2572.44;5.11093;0;0;0.1;0.01;0;2.65012;1;0;-1
0;3;-38.5132;0.472886;-85.5926;8.93259;2985.53;5.37104;0;0;0.1;0.01;0;2.4953;0;0;-1
0;4;-35.4891;20.6064;-84.3379;10.0373;4235.84;6.03528;0;0;0.1;0.01;0;3.05361;0;0;-1
0;5;-21.8021;-40.6281;-87.2579;9.31441;3384.97;5.60062;0;0;0.1;0.01;0;2.80819;0;0;-1
0;6;-20.5456;-22.4016;-85.933;8.72424;2781.45;5.24576;0;0;0.1;0.01;0;4.23956;0;0;-1
0;7;-21.0298;-8.41815;-84.5089;8.68751;2746.47;5.22367;0;0;0.1;0.01;0;4.1069;0;0;-1
0;8;-20.3025;10.7176;-85.9715;8.9461;2999.09;5.37916;0;0;0.1;0.01;0;3.39456;0;0;-1
0;9;-22.2958;28.7601;-84.0031;9.1147;3171.87;5.48054;0;0;0.1;0.01;0;3.81391;0;0;-1
0;10;-8.84437;-49.3344;-83.4182;10.3831;4688.89;6.24321;0;0;0.1;0.01;0;3.60858;0;0;-1
0;11;-7.09671;-30.7516;-87.1641;9.25676;3322.5;5.56596;0;0;0.1;0.01;0;4.75339;0;0;-1
0;12;-6.80614;-16.6166;-86.2506;8.92614;2979.06;5.36716;0;0;0.1;0.01;0;5.07848;0;0;-1
0;13;-6.59273;0.139274;-85.5164;10.1914;4433.94;6.12794;0;0;0.1;0.01;0;5.02053;0;0;-1
0;14;-7.56441;20.0313;-83.7587;10.1998;4444.92;6.13299;0;0;0.1;0.01;0;5.00831;0;0;-1
0;15;-7.67702;38.293;-83.9338;9.71243;3837.71;5.83994;0;0;0.1;0.01;0;3.32165;0;0;-1
0;16;7.44445;-40.87;-85.9511;10.0896;4302.4;6.06673;0;0;0.1;0.01;0;4.84768;0;0;-1
0;17;6.40276;-24.2342;-83.4865;10.0954;4309.82;6.07022;0;0;0.1;0.01;0;6.03818;0;0;-1
0;18;8.72537;-6.01744;-85.8347;8.76963;2825.09;5.27305;0;0;0.1;0.01;0;3.89397;0;0;-1
0;19;6.47914;10.9385;-86.7586;8.73724;2793.9;5.25358;0;0;0.1;0.01;0;2.66741;0;0;-1
0;20;7.75366;27.7551;-87.3932;8.84193;2895.54;5.31652;0;0;0.1;0.01;0;3.18335;0;0;-1
0;21;8.87651;45.3857;-83.5773;10.2972;4573.47;6.19156;0;0;0.1;0.01;0;3.802;0;0;-1
0;22;20.2418;-49.154;-84.4952;9.44966;3534.57;5.68194;0;0;0.1;0.01;0;3.23383;0;0;-1
0;23;23.3363;-34.0333;-86.6651;10.0918;4305.21;6.06805;0;0;0.1;0.01;0;5.12303;0;0;-1
0;24;20.6083;-15.7285;-84.9688;8.5;2572.44;5.11093;0;0;0.1;0.01;0;3.28161;1;0;-1
0;25;22.4796;3.45145;-86.8633;8.58914;2654.22;5.16453;0;0;0.1;0.01;0;3.5917;0;0;-1
0;26;23.633;20.3931;-84.4861;8.88816;2941.2;5.34432;0;0;0.1;0.01;0;4.47596;0;0;-1
0;27;23.5491;37.7414;-84.8626;9.8457;3997.87;5.92008;0;0;0.1;0.01;0;3.69262;0;0;-1
0;28;36.3515;-40.1918;-83.649;9.99562;4183.29;6.01022;0;0;0.1;0.01;0;3.76918;0;0;-1
0;29;35.4709;-25.5438;-86.1256;8.5;2572.44;5.11093;0;0;0.1;0.01;0;3.01551;1;0;-1
0;30;37.2092;-5.59643;-85.3022;10.2046;4451.19;6.13588;0;0;0.1;0.01;0;2.52933;0;0;-1
0;31;36.5584;10.8587;-83.9695;10.2277;4481.49;6.14977;0;0;0.1;0.01;0;3.93953;0;0;-1
0;32;37.5893;26.9807;-83.927;10.0868;4298.82;6.06505;0;0;0.1;0.01;0;4.35845;0;0;-1
0;33;-70.5277;-14.7649;-69.3765;9.3986;3477.59;5.65124;0;0;0.1;0.01;0;2.27981;0;0;-1
0;34;-57.8121;-39.4461;-71.3458;9.97762;4160.73;5.9994;0;0;0.1;0.01;0;3.10537;0;0;-1
0;35;-56.6684;-25.1883;-72.4934;9.30194;3371.39;5.59312;0;0;0.1;0.01;0;3.99619;0;0;-1
0;36;-54.7581;-8.89582;-70.8532;8.5;2572.44;5.11093;0;0;0.1;0.01;0;4.50407;1;0;-1
0;37;-56.2927;9.85667;-69.7953;10.3383;4628.45;6.21627;0;0;0.1;0.01;0;3.50748;0;0;-1
0;38;-57.7911;28.4754;-71.5058;8.5;2572.44;5.11093;0;0;0.1;0.01;0;2.15643;1;0;-1
0;39;-54.6542;46.9027;-68.5817;9.52835;3623.61;5.72926;0;0;0.1;0.01;0;3.23974;0;0;-1
0;40;-41.548;-51.3145;-72.6758;9.08313;3139.03;5.46155;0;0;0.1;0.01;0;2.11194;0;0;-1
0;41;-41.3985;-30.4538;-70.9856;9.44447;3528.75;5.67882;0;0;0.1;0.01;0;5.73716;0;0;-1
0;42;-42.2318;-16.6243;-69.6692;10.4347;4759.14;6.27423;0;0;0.1;0.01;0;6.85618;0;0;-1
```

Figure 3.3: Cells' Output Format

Thus, regarding the phase's integer code, we map it to the appropriate category among the three general ones; "Alive", "Apoptotic" and "Necrotic" and count the number of agents/cells included in each one of the aforementioned categories for each timestamp. Obviously, at the first timepoint (timepoint = 0) all the agents/cells belong to the state "Live", so, all the cells have phase that corresponds to the "Alive" category. Therefore, at timepoint = 0 we meet the total amount of starting agents/cells.

```
0 "Ki67_positive_premitotic"
1 "Ki67_positive_postmitotic"
2 "Ki67_positive"
3 "Ki67_negative"
4 "G0G1_phase"
5 "G0_phase"
6 "G1_phase"
7 "G1a_phase"
8 "G1b_phase"
9 "G1c_phase"
10 "S_phase"
11 "G2M_phase"
12 "G2_phase"
13 "M_phase"
14 "live"
100 "apoptotic"
101 "necrotic_swelling"
102 "necrotic_lysed"
103 "necrotic"
104 "debris"
```

Figure 3.4: Cell Cycle Map

3.5 Message Format

At the next step, we needed an identifier which would uniquely determine a specific simulation. For this reason, we used the pid (process identifier) using the usual function *getpid()*. The whole processing that takes place at the consumer, is done according to these pids so as to be able to separate the simulations from each other. At this point we need to mention that the pid is recognizable only to processes that run in the same machine. Otherwise, in other machines, it is just a number and they cannot be managed, using their pids. So, we need some information to be able to know in which simulation each pid corresponds. For this reason, the second field – after pid - we need to send, is the path, which includes the information about the exact location of a specific simulation, as well as, the number of the specific simulations of a concrete example. In this way, we cover the case of different simulations, from different examples, but with the same run's folder number, are simultaneously running, without the user's confusing.

Finally, concatenating all the necessary information, meaning pid, path, timepoint, as well as, alive, apoptotic and necrotic cells' number we counted previously, we are ready to send the message depicted in Figure 3.5 to Kafka.

pid	path	timepoint	aliveNO	apoptoticNO	necroticNO

Figure 3.5: Message Format

3.6 Message Consuming

The next step is message consumption. As we already mentioned, the tool, we used to consume the messages, is Confluent's Kafka Python. The first thing that we must do is to start running the consumer instances, with each one of them having as argument the partition's id from which they will pull messages. So, it is concluded that the number of partitions and the number of consumers coincide. The very first thing we do, after decoding the pulled message is to create a dictionary with values, the discrete elements of the message and inserts it into the desirable database.

Each partition is a well-defined queue that keeps messages following the FIFO (First In First Out) structure. The content of each partition is not only messages coming from different simulations, namely, different pids, but also from probably different timepoints, since not necessarily all simulations started simultaneously or even that they have the same output interval parameter. For example, a specific partition may include data coming from simulation X and simulation Y, with the X simulation having as output interval 30 minutes and Y 40 minutes. Also, the last received message from X may be at timepoint 120 and from Y at timepoint 160. For this reason, when we consume the messages, we do a linear interpolation among the last valid and the current timepoint for this specific simulation generating intermediate values.

All the different coming simulations are stored in a dataframe called "Simulations" (see Table 3.1). This data structure has as index the "Time" (for the needs of the example we depict the Time column with step 30) and as name columns all the different simulations/pids the specific consumer manages. Each pid column carries its personal alive statistics located at the respective time index.

	PID_0	PID_1	PID_2	...	PID_N
Time					
0	$Alive_{0,0}$	$Alive_{1,0}$	$Alive_{2,0}$	...	$Alive_{N,0}$
30	$Alive_{0,30}$	$Alive_{1,30}$	$Alive_{2,30}$	...	$Alive_{N,30}$
60	$Alive_{0,60}$	$Alive_{1,60}$	$Alive_{2,60}$	...	$Alive_{N,60}$
90	$Alive_{0,90}$	$Alive_{1,90}$	$Alive_{2,90}$	...	$Alive_{N,90}$
...	...	...	...	...	...
1440	$Alive_{0,1440}$	$Alive_{1,1440}$	$Alive_{2,1440}$	...	$Alive_{N,1440}$

Table 3.1: Simulations Dataframe

At this point we need to mention that the statistics are normalized so as they can be comparable to the cells of another simulation which has different number of starting agents. For example, the simulation X may have 1000 starting agents instead of the simulation Y which has 2000, a fact that makes the two (2) simulations incomparable to each other and all the following computations wrong. For this reason, we normalize each statistic with the number of its simulation's starting agents. Thus, all the simulations the consumer handles, have "1" as alive cells number at "0" timepoint.

In addition to the "Simulations" dataframe, we have another one data structure, too, the "Nominated" one. At Table 3.2 we provide a numerical example of how this table looks during the procedure. Initially, the table's values are zero (0). Each row of the "Nominated" dataframe contains information about a simulation's pid, as well as, its score. To be more specific, the DownTrend represents the last numerical distance among a simulation X and the threshold, a simulation that is used as a ruler to decide if a simulation deserves to live or not. The CountDown represents the "chances to life".

index	PID	DownTrend	CountDown
0	Pid4	0.5	1
1	Pid2	0.25	2
2	Pid100	0.4	4
...	...	...	...
N	Pid5	1	5

Table 3.2: Nominated Dataframe

Finally, the last data structure we use is a list named *top_k* which includes every moment the best *k* pids/simulations of the measurements.

Having mentioned all the necessary information and tools that were used, we can now pass to the algorithm's analysis and explanation.

3.7 Algorithm

First of all, the consumer is divided into the following cases:

- If (timepoint == 0)
- If (timepoint > 0)

Considering the first case, if the received message, at the time's field has the value "0", that means that this is the first received message for this specific simulation. So, we create a new column in the "Simulations" dataframe with column name, this simulation's pid and we append the normalized number of alive cells at the first (1^{st}) row (time = 0). Also, we append it as a new row to the "Nominated" dataframe with the DownTrend and CountDown be initially zero (0). Subsequently, we check if the top_*k* list is full or not. If it is not, then we insert the pid to the top_*k* list. The first *k* different pids/simulations that come, appended to the top_*k* list.

So, in this case we should not apply the predicting algorithm since, at zero (0) timepoint all cells are alive in every simulation. Also, if the total amount of simulations that the consumer handles is less or equal to *k*, then the algorithm is not applied again.

Now, considering the second case, for every received message with timepoint greater than the initial one and under the circumstance that top_*k* list is full, we apply the algorithm so as to judge if the simulation should be added to the top_*k* list replacing the threshold, get killed or stay alive.

The first thing we should do is to set the threshold, namely, the worst out of the best *k* simulations listed in the top_*k*. To achieve that, we should do two (2) procedures. The first one is to find out the minimum last received valid timepoint among the top_*k* ones. The second one is to find the worst simulation, namely, the simulation with the most alive cells at the timestamp we found as minimum in the previous procedure. It is important to do our calculations at the same timestamps, otherwise the measurements have no sense (see Table 3.3).

	PID_0	PID_1	PID_2	...	PID_N
Time					
0	$Alive_{0,0}$	$Alive_{1,0}$	$Alive_{2,0}$	...	$Alive_{N,0}$
30	$Alive_{0,30}$	$Alive_{1,30}$	$Alive_{2,30}$	...	$Alive_{N,30}$
60		$Alive_{1,60}$	$Alive_{2,60}$	...	$Alive_{N,60}$
90		$Alive_{1,90}$		...	$Alive_{N,90}$
...		...		...	...
1440				...	

Table 3.3: Maximum Common Timepoint

Consequently, knowing that the top_*k* list is full and ignoring the cases where time is zero (0), we must check if the threshold has already generated value for the time of the current simulation.

If yes, then we compare at this specific timepoint the number of the alive cells of the current simulation with the ones of the threshold. If the simulation's alive cells are less than the threshold's, then we throw out of the top_*k* list the until now threshold and we replace it with the current simulation's pid. Now, the new threshold is the current simulation. Otherwise, if the new one is not better than the worst out of the best one, then we check its score. Concretely, if the DownTrend, namely, the downward trend of this specific simulation in relation with the threshold, is greater than the one in a previous timepoint, that means that the simulation's course is rising and for this reason, we increase its CountDown by one, updating the Nominated dataframe not only for the CountDown, but for the new numerical distance as well. Afterwards, we check if it is high time the simulation should be killed. Specifically, if the CountDown of the under-examination simulation reached the *5*, that means that the life chances, namely, the tolerance for this specific simulation, finished and for this reason it should be killed.

The killing part varies according to the design. To be more specific, if the whole procedure takes place in a local server without more than ones different ips interfere, then we can simply kill the simulation regarding to its pid calling the *kill* function which is an attribute of the os module in Python. If the simulations are distributed into different machines, as well as, the consumers, then we should transfer the killing information through Kafka and by the time the message is consumed, then a flag is raised and the under execution PhysiBoSS instance gets killed. Finally, if the whole procedure takes place in a cluster or grid computer, then we can just kill this specific simulation's jobid by calling the appropriate function depending on the used resource manager (for us was

TORQUE as resource manager and qdel as the killing command) avoiding thus, the use of extra Kafka resources and as a result the whole system's complexity to be increased.

A question that emerges from the whole procedure is why we decided to show some tolerance to the potential non-hopeful simulations inducing as a result time delay.

The reason why we did this, is lying on several factors. One of the most important, which can nowhere and never can be disregarded is the noise. For example, the simulation depicted on Figure 3.6 has a little bit of an unpredicted response through time since it has great fluctuation. If we compare this one to the one depicted on Figure 3.7, then it will be killed for sure and there is no reason for CountDown to exist. However, if we compare it with the Figure 3.8, totally ignoring the tolerance, then at the timepoint 200 Figure 3.8 will beat the Figure 3.6. However, finally, the one that should have survived is the Figure 3.6. Considering the tolerance, then we have the desired results.

To sum up, if we avoid this tolerance, we kill simulations earlier, but with the cost that maybe a dead simulation was, finally, better than one of the k ones which survived. So, we prefer to sacrifice time instead of better results. After different experiments, we decided that countdown 5 is a good tolerance for average simulations. Of course, this can be modified considering the simulations' nature and how competitive they are to each other.

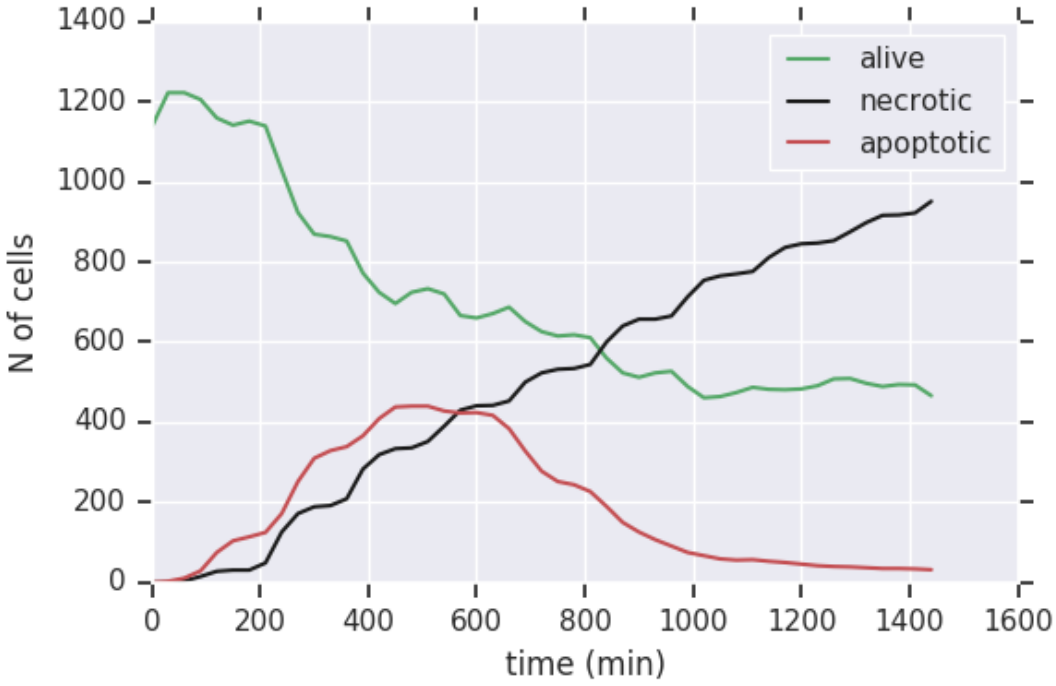

Figure 3.6: pulse150_oxy0

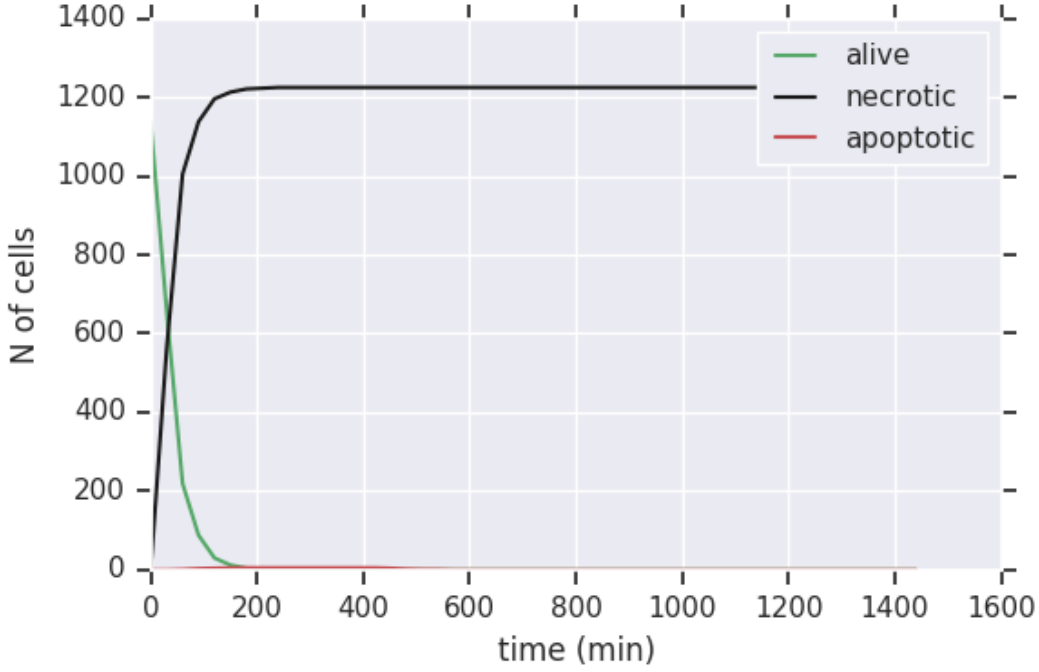

Figure 3.7: pule150_oxy0.1

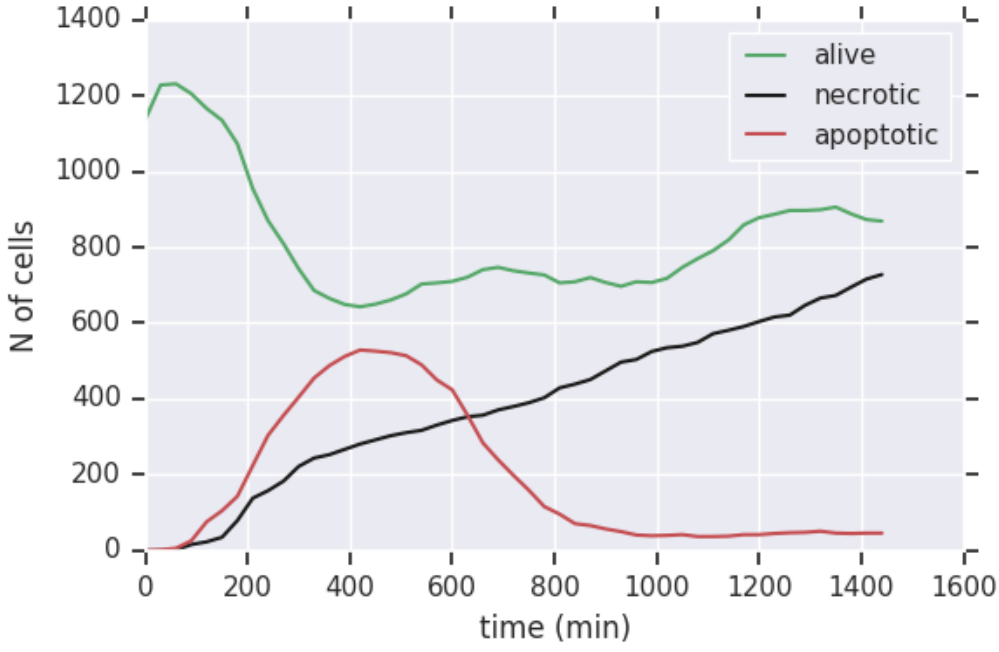

Figure 3.8: Pulse100_oxy0

Back to our algorithm, if it is decided that the simulation must be killed, then we drop any existed information about it in the database and the "Simulations" dataframe, as well as, we generate a .txt file in which information about the killed simulations is included.

The procedure is repeated until the consumer has pulled all the messages. Figure 3.9 contains the pseudocode of the consuming procedure.

```
BEGIN

k := 5

Create a mongo client
Create a db and make a connection to it

Pull message
Decode message

pid := msg[0]
path := msg[1]
alive := msg[2]
apoptotic := msg[3]
necrotic := msg[4]

Store data in mongoDB

IF (time == 0) THEN
    norm_value := alive
    alive := alive/norm_value
    Create new column in simulations Dataframe, named with sim's PID
    Append data in simulations Dataframe
    Create new row in nominated Dataframe
    IF (top_k NOT FULL) THEN
        top_k.append(pid)
    ENDIF
    CONTINUE consuming
ENDIF

# For every other timepoint > 0
alive := alive/norm_value
Interpolate the intermediate values
IF (top_k is FULL) THEN
    FOR simulation in top_k
        minimum := find the maximum last common timepoint
    ENDFOR
    FOR simulation in top_k
        threshold := find the worst of the top_k
        swap positions(top_k[0], threshold)
    ENDFOR
    IF (minimum > 0 AND (pid NOT IN top_k)) THEN
        IF (simulations.at[time,threshold] EXISTS) THEN
            IF (alive < simulations.at[time,threshold]) THEN
                top_k[0] := pid
            ELSE
                IF (nominated.at[pid,DownTrend] < round(alive - simulations.at[time,threshold])) THEN
                    nominated.at[pid, CountDown] += 1
                nominated.at[pid, DownTrend] := round(alive - simulations.at[time,threshold])
                ENDIF
            ENDIF
        ELSE
            maxTime := simulations[threshold].last_valid_index()
            IF (simulations.at[maxTime,pid] < simulations.at[maxTime,threshold]) THEN
                top_k[0] := pid
            ELSE
                IF (nominated.at[pid,DownTrend] < round(alive - simulations.at[maxTime,threshold])) THEN
                    nominated.at[pid, CountDown] += 1
                nominated.at[pid, DownTrend] := round(alive - simulations.at[maxTime,threshold])
                ENDIF
            ENDIF
        ENDIF
        IF (nominated.at[pid,CountDown] >= 5) THEN
            kill simulation pid
            delete info in mongoDB
            delete simulations[pid]
            write simulations info in a file
        ENDIF
ENDIF

END
```

Figure 3.9: Algorithm's Pseudocode

After that, the last module in the whole system, *TOP_K* module, just proclaims the best k out of all the received locally top_*k*'s taking into account the last alive's value. If there is overlapping among some simulations at the last timepoint's value, then we can either permit *k* to be extended or randomly choose just *k* among the best, depending on the user's need. Also, *TOP_K* module supports the opportunity of a report file creation, in which there is included information about the survival simulations, as well as, survival simulations representation with a figure (e.g. Figure 3.6), if the user gives the appropriate arguments when calling *TOP_K* (*python3 top_k_global.py -f=report.txt -p*)

4. Experimental Results

What we, finally, expect from our whole design is:

- *Effectiveness*, regarding the simulations' survival rate
- *Time performance*, regarding the average killing timepoint
- *Distributed* functionality
- *Parallelization computing*

To verify the effectiveness, as well as, to test the whole performance of our design, we brought about an amount of experiments. The experiments took place locally and remotely, as well, so as to confirm its functionality gives promising results not only for a few simulations running in a local computer, but for high volume data running distributed into different servers, too.

At this point we need to mention that for the needs of the experimental part, we used a grid computer. This execution platform consisted of a local server and forty-four (44) independent to each other nodes each one of them consisted of 4 processors. So, the maximum number of simulations that we could simultaneously run was:

$$4\ ppn \times 44\ nodes\ = 176\ real - time\ distibuted\ running\ simulations,$$

where ppn stands for processor per node.

The local server communicates with the nodes using TORQUE [39] and openMPI. Thus, the local server has the ability to start, hold, cancel and monitor jobs. Some of the experiments conducted in a computer grid and some others in a local computer using two (2) virtual machines – with different IPs - one (1) for the producing and one (1) for the consuming part. Also, all the measurements were done under the circumstance that: $\boldsymbol{k} = \mathbf{5}$ and the number of the simultaneously running consumers were three (3).

To verify the functionality of the design we had to check it in a varying amount of simulations and consider both the average and the extreme cases, too. In this way, we divided the experiments into three (3) general categories according to the simulations' nature; average, promising and non-promising.

4.1 Use Case 1: Average Simulations

For our first (1st) scenario, we tested the system's performance for simulations that their number of alive cells have average response through time, like the one depicted in Figure 4.1.

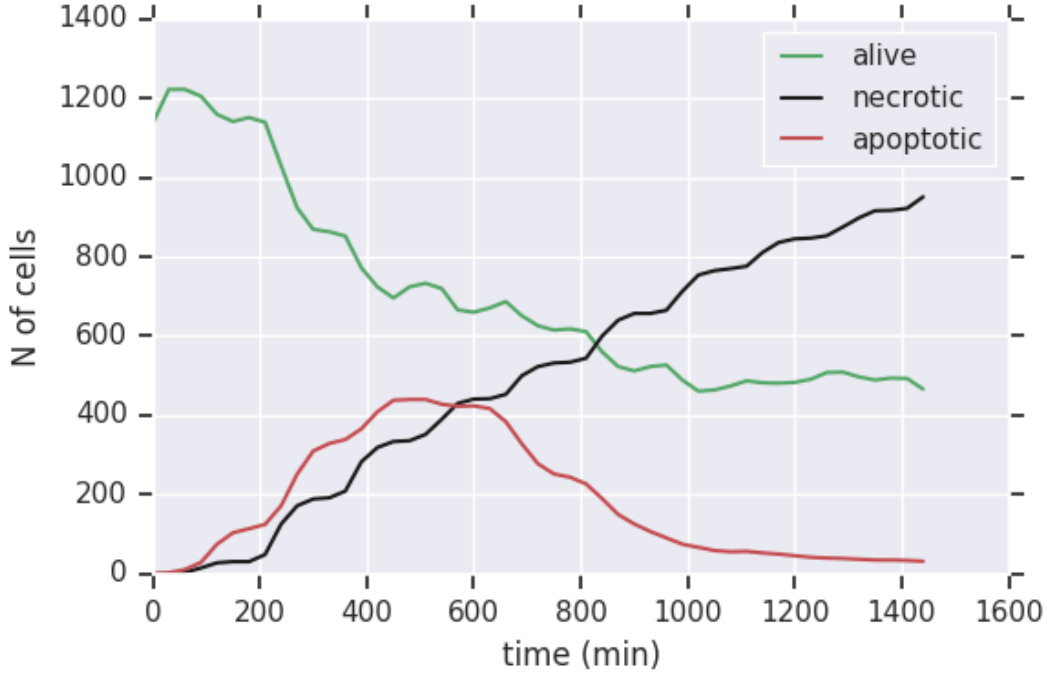

Figure 4.1: Pulse150_ TNFconc0.5_oxy0

For the needs of this group of experiments we used simulations from different examples, starting with a random set of parameters. Also, we made sure that the *output_interval* parameter, namely, the parameter that defines the sampling time, differs among simulations so as to ensure that it does efficiently covered this case, too. In addition to that, we took measurements for two (2) well-defined examples, too, *pulse25_600_conc* and *pulse5-1400_conc* in which the changeable parameters were the *time_add_tnf*, which defines the interval among pulses and the *tnf_concentration*, which constitutes the dosage of the medicine at each pulse. The statistics that we received from them were the ones that are presented in Table 4. 1.

Examples	**# Simulations**	**# Consumers**	**# Survived**	**# Dead**	**# Top *K* (global)**	**Average Killing Time**
random	230	3	5/113 5/54 5/63	215	**5**	20.85%
pulse5-1400_conc	176	3	30	146	**5**	22.2%
pulse25-600_conc	96	3	5/30 6/31 5/35	80	**5**	25.8$\bar{3}$%

Table 4. 1: Average Simulations Statistics

From the measurements we can conclude that for the average cases, the algorithm's efficiency is very high, since for all the above examples the final number of simulations was equal to the *k*. As for the time, the average killing timepoint was, approximately, 300 - 380 minutes and considering to the whole procedure lasts 1440 minutes, this is a

very satisfying killing percentage, since the average simulation gets killed before the half of the whole procedure.

4.2 Use Case 2: Promising Simulations

There are cases that a specific group of simulations modify only one parameter in a range where the step is very small in order to better capture transition effects and finally find where the threshold is (see Figure 4.2).

A very important simulation's parameter is *oxygen_necrotic* which is an environmental cause of necrosis happening in real life. Knowing that the zero (0) value means that there is no way to go to necrosis by lack of oxygen and that only values below two (2) give proliferative outcomes, we created a clustering of one thousand (1000) ready to run simulations (*example spheroid TNF pulse150 oxy*) with all the parameters being the same and the only changeable parameter being the *oxygen_necrotic* in a range of 0 to 1 with step 0.001.

Thus, using three (3) consumers and randomly* allocating the simulations into three (3) partitions, we received the following results for varying cases of *CountDown*:

CountDown	# Simulations	# Consumers	# Survived	# Dead	# Top *K* (global)	Average Killing Time
5	1000	3	989	11	979	$33.7\overline{12}\%$
3	1000	3	**543**	457	**543**	9%
2	1000	3	**416**	584	**416**	5.2%
1	1000	3	**248**	752	**248**	2.75%

Table 4.2: example_spheroid_TNF_pulse150_oxy Statistics

The above table gathers the results we received for *CountDown* giving the values 5, 3, 2 and 1. As we can see, in the case of *CountDown* being 5 the simulations that survived are 989 out of the initial 1000 simulations, which means than only the 11 died. However, we can see that neither the global module could do something significantly better, since it ignored only the 10 of the 989. The average timepoint that these simulations were killed was $485.\overline{45}$ minutes, which means that each simulation got killed at the $33.7\overline{12}\%$ of the whole procedure.

Thus, we concluded that we need a stricter tolerance than the average used one, 5. So, we repeated the same experiment for *CountDown* being 3. In this case the results are very interesting. First of all, we can notice that the difference between the dead ones with tolerance 5 and the dead ones with tolerance 3 is significant, since this time $547/1000$ got killed instead of the 11 of the previous measurement. Finally, the global Top *K* module is forced to surpass the *k* since at the last timepoint much more simulations ended up with the same alive cells' value. The remarkable part is that all the simulations that globally survived are exactly the summary of the locally survived ones with the average deciding time being approximately at 130 minutes out of the 1440 minutes, which is the whole simulation's duration. Similar were the results, while we were decreasing the tolerance. In this point, we need to mention that we applied a recursive procedure in order to find the last timestamp for which the simulations are

* We considered to maintain the same allocation for all the different values of CountDown to have a clear view about the results.

overlapped and by extension the threshold. It is important we mention that, we found out that 242 out of the 1000 simulations were totally identical considering the number of the alive cells, since from the second timepoint and after all the cells fell into necrosis. Below we cite the statistics of another example consisted of *1200* simulations in which we vary the *oxygen_necrotic* (0, 0.9, 0.1), the *time_add_*tnf (100, 300, 50), the duration*_add_tnf* (5, 15, 5) and the *tnf_concentration* (0.1, 0.8, 0.1) parameters.

Again, the simulations are randomly allocated into three (3) partitions:

CountDown	# Simulations	# Consumers	# Survived	# Dead	# Top *K* (global)	Average Killing Time
5	1200	3	305/335 430/445 345/420	120	**1080**	22.17%
2	1200	3	304/335 430/445 345/420	121	**1079**	10.49%
1	1200	3	110/335 105/445 81/420	904	**296**	3.05%

Table 4.3: example_spheroid_TNF_pulse100-300_dur_conc_oxy Statistics

Most of the simulations' response for both examples were looking like Figure 4.2. We can conclude that from the second timepoint and ahead, most cells fall into necrosis statement.

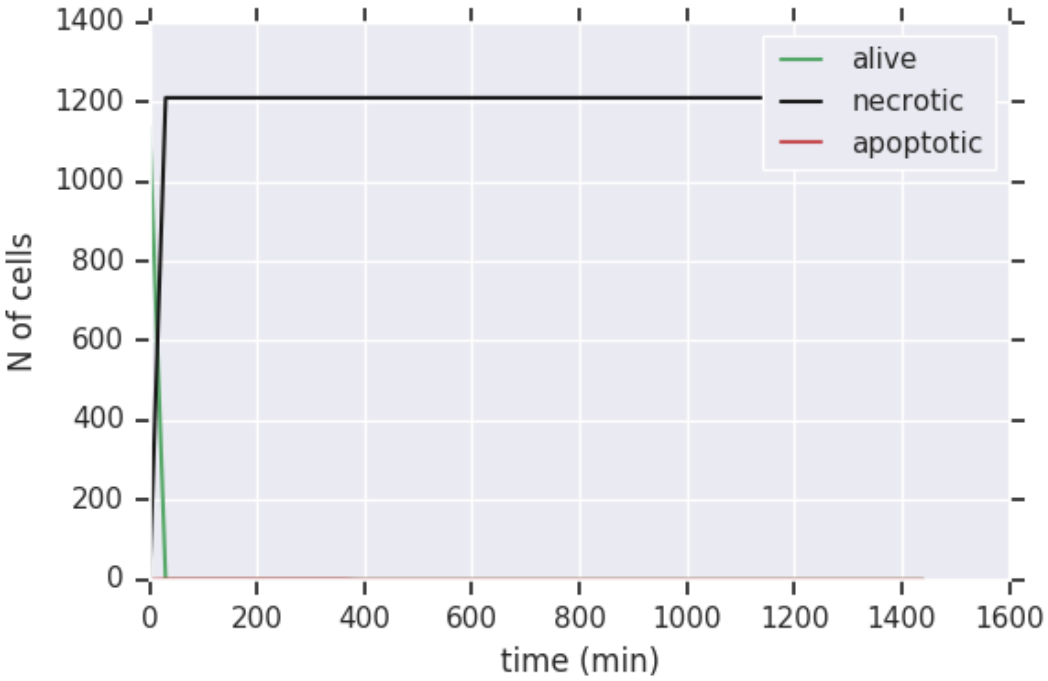

Figure 4.2: pulse150_oxy_0.5

Generally, in such cases where we try to find out a threshold in a very tight range, the more we decrease the tolerance, the better results we get. So, we can totally ignore it to save time and limit the results. Another solution would be instead of using the alive cells as ruler at the global Top *K*, we could use the number of apoptotic or necrotic cells

if we care about them, but this depends on what the user needs. The conclusion in that scenarios is that, inevitably the ones we kill are also good, but they did not survive because better ones existed.

4.3 Use Case 3: Non-Promising Simulations

In this scenario, we do the hypothesis that all simulations are bad. According to our algorithm's current design some of them and specifically *k*, will survive since, right now the algorithm makes sure to return the top *k* out of all the received ones. So, for the case that all simulations of a specific classification are going to have a bad performance, namely, non-promising ones with response as the one depicted at Figure 4.3, then the best *k* of the worst inevitably will survive.
For this reason, we suggest that algorithm has an initial threshold, so as to finally none non-promising simulation stays alive even if it was the best out of the worst.
Regarding the standard threshold, that could be either a simulation loaded from the memory or the 1 (the number of starting agents) to every timepoint. In both cases we have a good time saving solution.
In the specific case that the user needs to keep some of the non-hopeful ones, because they want to depict the general response through time of a cluster, then they can ignore the standard threshold.

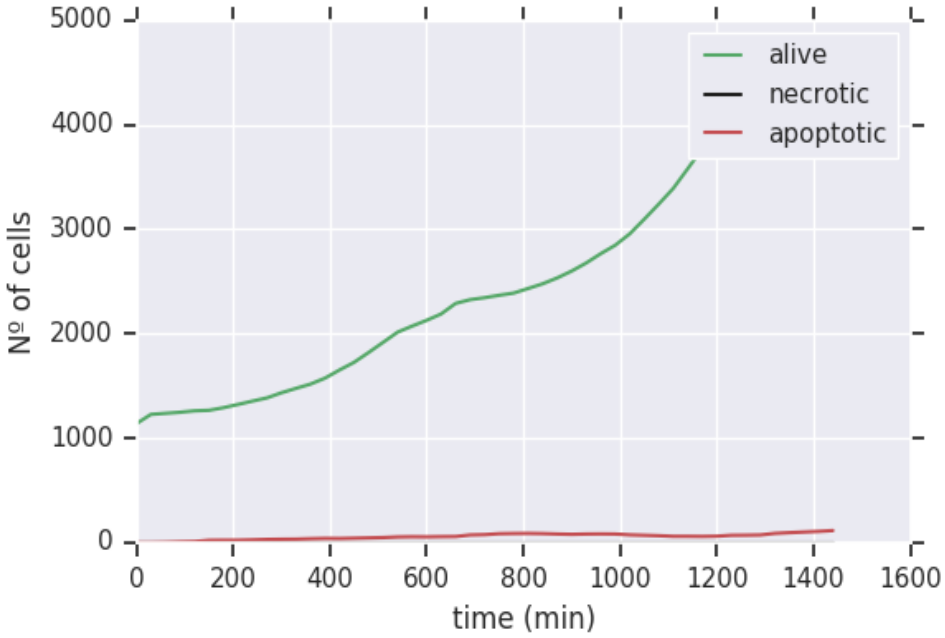

Figure 4.3: nopulse

Concluding with the experimental part, we need to highlight that the results depend on how competitive simulations' nature is, but, generally, we can express that the efficiency is good, thinking that all the non-hopeful simulations are killed and specifically before the half of the whole simulation's duration.

5. Conclusion – Future Work

5.1 Conclusion

Concluding, in this diploma thesis we designed an algorithm that collects through Apache Kafka the statistics of all real time running simulations in a parallel way. The parallelization level depends on the number of running consumers. Every consumer's instance continuously compares the simulations one to another attending to keep locally the top *k* ones at all moments. The ones that are excluded from the top *k* list do not seem to be promising enough to be completed and thus they get killed. After all consumers are completed, they send their top *k* list to the global Top *k* module which compares the individual received top *k* and decides the final top *k* simulations regarding on the alive cells' number at the last timepoint. In case that there is an overlapping among simulations, then the global Top *k* can either keep randomly *k* ones or permit *k* to be increased as much as it is demanded to cover all and not let one simulation be excluded. The design supports both local and remote communication.

We verified its performance, by running the algorithm repetitively for an adequate number of different clustered examples (approximately 80 GB data), trying to include good, average and bad simulations, too, so as to test it to every possible combination of simulations and to take into account extreme cases, like all real time running simulations to be promising.
The results were hopeful for the final number of surviving simulations considering the initial number, as well as for the average time that the simulations get killed, achieving to gather results before each simulation reaches the 40% of the whole procedure.

5.2 Future Work

The performance as well as the complexity of the current algorithm could possibly be improved under some modifications. First of all, we could store all the real-time from N running PhysiBoSS instances generated data into a buffer/queue and this buffer/queue would be considered to be the only producer of the whole system. Thus, we reduce currently used Kafka resources and specifically producers, and by extension the whole complexity from N to 1. As for the messages' allocation from the producer to the partitions, instead of it being random, a more sophisticated implementation (e.g. Hashing) could be used. Furthermore, an essential improvement would be that the local top *k* instances continuously report the *k* best simulations to the global Top *k* module (real-time top-*k* monitoring) which would also have the permission to kill the simulations that locally belong to the top-k list, but not globally. As a result, we would manage to reduce the overhead and increase the performance. In this way, we could ignore the database, that we use right now. Finally, probably the most important improvement would be to embed the whole system inside the Apache Flink framework, which is specifically designed to permit distributed processes, considering that it offers stateful computations over unbounded and bounded data streams with high performance and speed.

Bibliography

[1] G. Letort, A. Montagud, G. Stoll, R. Heiland, E. Barillot, P. Macklin, A. Zinovyev, L. Calzone, *PhysiBoSS: a multi-scale agent-based modelling framework integrating physical dimension and cell signalling* (2018)

[2] G. Stoll, B. Caron, E. Viara, A. Dugourd, A. Zinovev, A. Naldi, G. Kroemer, E. Barillot, L. Calzone*, MaBoSS 2.0: an environment for stochastic Boolean modeling* (2017)

[3] A. Ghaffarizadeh, R. Heiland, S. H. Friedman, S. M. Mumenthaler, P. Macklin, *PhysiCell: An open source physics-based cell simulator for 3-D multicellular systems* (2018)

[4] A. Ghaffarizadeh, S. H. Friedman, P. Macklin *BioFVM: an efficient, parallelized diffusive transport solver for 3-D biological simulations* (2015)

[5] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, A. Schuster, Prediction – Based Geometric Monitoring Over Distributed Data Streams (2012)

[6] B. Babcock, C. Olston, Distributed Top – K Monitoring (2003)

[7] https://maboss.curie.fr/

[8] http://physicell.org/

[9] https://github.com/sysbio-curie/PhysiBoSS

[10] https://github.com/gletort/PhysiBoSS/wiki/Parameters

[11] https://www.slideshare.net/rahuldausa/introduction-to-kafka-and-zookeeper

[12] https://sookocheff.com/post/kafka/kafka-in-a-nutshell/

[13] https://www.tutorialspoint.com/apache_kafka/index.htm

[14] https://kafka.apache.org/quickstart

[15] https://www.tutorialspoint.com/apache_kafka/apache_kafka_cluster_architecture.htm

[16] https://github.com/mfontanini/cppkafka

[17] https://github.com/edenhill/librdkafka

[18] https://github.com/confluentinc/confluent-kafka-python

[19] https://timber.io/blog/hello-world-in-kafka-using-python/

[20] https://www.programcreek.com/python/example/98440/kafka.KafkaConsumer

[21] https://buildmedia.readthedocs.org/media/pdf/kafka-python/master/kafka-python.pdf

[22] https://docs.confluent.io/5.0.0/clients/confluent-kafka-python/index.html

[23] https://towardsdatascience.com/kafka-python-explained-in-10-lines-of-code-800e3e07dad1

[24] https://kafka-python.readthedocs.io/en/master/index.html

[25] https://towardsdatascience.com/getting-started-with-apache-kafka-in-python-604b3250aa05

[26] https://matplotlib.org/3.1.1/tutorials/introductory/pyplot.html

[27] https://pandas.pydata.org/

[28] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

[29] https://docs.python.org/2/library/xml.dom.minidom.html

[30] http://epydoc.sourceforge.net/stdlib/xml.dom.minidom.Document-class.html

[31] https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

[32] https://www.w3schools.com/python/python_mongodb_getstarted.asp

[33] https://towardsdatascience.com/the-complete-guide-to-time-series-analysis-and-forecasting-70d476bfe775
[34] https://www.youtube.com/watch?v=bn8rVBuIcFg
[35] https://www.youtube.com/watch?v=VYpAodcdFfA
[36] https://www.youtube.com/watch?v=e8Yw4alG16Q
[37] https://hpcc.usc.edu/support/documentation/running-a-job-on-the-hpcc-cluster-using-pbs/
[38] https://www.jlab.org/hpc/PBS/qsub.html
[39] http://docs.adaptivecomputing.com/torque/6-0-2/adminGuide/help.htm#topics/torque/2-jobs/multiJobSubmission.htm
[40] https://slurm.schedmd.com/sbatch.html
[41] https://www.altair.com/pdfs/pbsworks/PBSUserGuide18.2.pdf
[42] https://kb.iu.edu/d/avmy
[43] http://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables
[44] https://www.tutorialspoint.com/cplusplus/index.htm
[45] http://www.cplusplus.com/doc/tutorial/