Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 349 (2020) 119–134

www.elsevier.com/locate/entcs

A Microservice Approach for a Cellular Automata Parallel Programming Environment

Aurelio Vivas [1,2] John Sanabria[3]

School of Systems Engineering and Computing
Universidad del Valle
Cali, Colombia

Abstract

This paper presents an architecture for a parallel programming environment designed to assist the process of writing parallel programs. The proposed architecture allows the integration of several tools such as parallel programming skeletons and parallelizing compilers, among others. On the one hand, parallel programming skeletons simplify the design of parallel algorithms. On the other hand, parallelizing compilers rely on compiler capabilities to translate sequential programs into ones capable of using multiple processing units (CPU/GPU) found in personal computers (PCs). The technological solution presented in this work follows a microservices-oriented approach. This approach eases the integration of new parallel programming tools in the existing deployment through well-known protocols and existing services. We show how the aforementioned parallel programming tools were integrated to support the coding and parallelization of a general purpose cellular automata. Since the parallelization rely on the capabilities underlying to each tool, the focus of this work is not to deal with an efficient cellular automata parallelization. Instead, a promising microservice approach to the development of an extensible and scalable parallel programming environment is presented.

Keywords: parallel programming environment, cellular automata, microservices, containers, docker, kubernetes

1 Introduction

Several tools have been developed independently to assist in the process of writing parallel programs. However, some tools are unknown by the scientific community or are not considered, given their difficulty of use. Existing parallel programming environments have been developed with a particular purpose in mind. Therefore, we have identified a lack of tools capable of integrating diverse parallel approaches

[1] CIBioFi Project, funded by the Colombian Science, Technology and Innovation Fund - General Royalties System (Fondo CTeI- SGR), Gobernación del Valle del Cauca, and COLCIENCIAS, under contract No. BPIN 2013000100007

[2] Email: aurelio.vivas@correounivalle.edu.co

[3] Email: john.sanabria@correounivalle.edu.co

https://doi.org/10.1016/j.entcs.2020.02.016

which are mostly developed under heterogeneous systems (e.g. operating systems and programming languages).

To tackle the integration problem, we follow a virtualization approach via containers, a.k.a. operating-system-level-virtualization. Containers allow the deployment of self-contained computational environments. In this work, three containers were deployed. One deploys a parallel programming skeleton tool, the second one a parallelizing compiler and the last one deploys an on-line code editor to be supported for the aforementioned tools.

In order to enable the integration of these containers a web service module was deployed in each container. This web service module allows us to expose each container's functionality as a microservice. Therefore, the microservice approach enables the integration and extensibility of new capabilities in the proposed tool.

The outline of this paper is as follows: Section 2 is dedicated to introducing concepts relevant to this paper, such as containers, microservices and cellular automata. Section 3 describes similar studies to the work presented in this paper. Section 4 presents the proposed service-oriented architecture for developing parallel programs. Section 5 presents how the Game of Life is modeled with the proposed framework. Section 6, presents the microservice approach evaluation. Lastly, Section 7 describes conclusions and future work.

2 Background

2.1 Parallel Programming

In the last fifteen years, computer architectures have developed from mono processors (single core) to multi-core systems in order to overcome the physical limitations imposed by increasing the number of transistors in a single processing unit. The number of cores per processor has increased and the computer clock frequency has also decreased in order to reduce power consumption and heat generation. This change in processor design means that old sequential programs have no performance benefit from new multi-core systems. In order to deal with these new architectures, the parallel programming paradigm emerged as an approach which requires developers to identify in their programs units of work that can be performed in parallel by multiple processing units (CPUs, GPUs, etc). However, taking into account parallel units of work has become a new barrier considering our legacy of sequential thinking. This is the major reason why great variety of tools with different level of abstractions are developed to help users with parallel programming. Among the tools developed for this purpose are; parallel programming languages (e.g.High Performance Fortran), programming language libraries (PThreads, CUDA, MPI, OpenMP and OpenACC), automatic parallelizers, parallel programming environments (e.g. Parallware [2]), and solvers [4] (e.g. OpenFOAM, ParaView). The work described in this paper focuses on parallel programming environments.

[4] Solvers hide the parallelization details in solving problems that follow specific parallelism patterns.

2.2 Container

Containers are known as a lightweight virtualization approach that attempts to solve the problem of building and deploying software across multiple platforms and environments. They pack applications and their dependencies into standard units of software (containers) that are; portable, easy to build and deploy, have less storage requirements and low runtime overhead [8] [18, p. 1]. Containers can be moved from a desktop environment to the cloud and back to the physical servers easily and quickly [9]. Benefits offered by containers have met the needs of different areas, such as: cloud computing, high performance computing, reproducible research, software development, among others. People build single container applications using a container runtime such as Docker or Singularity. Applications such as microservices, consisting of several containers are deployed in distributed systems through container orchestration technologies such as: Docker Swarm, Kubernetes and Docker Compose. In particular, Kubernetes orchestration includes scheduling and scaling of a set of containers. The basic unit of scheduling in Kubernetes is called Pod. A Pod is a collection of one or more containers which share the same network interface [24, p. 3]. Usually, developers place services that are closely related in the same Pod.

2.3 Microservice Architectural Style

James Lewis and Martin Fowler describe microservices as follows: "the microservice architectural style is an approach used to develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms" [4]. Microservices architectures gain a lot of popularity due to the ever-increasing complexity of maintaining and scaling monolithic applications. They are developed under the concept of "do one thing and do it well", i.e., each microservice must be focused on a single business capability which can be delivered and updated independently [5]. Microservices may cooperate in order to provide more complex and elaborate functionalities. Cooperation can be established under the orchestration or choreography scheme.

Orchestration means that a central service is in charge of coordinating other services' activities. The central service receives a request from either a user or an external service. It handles the request by making a new request to one or more of the services for which it is responsible. After receiving a response, the central service returns a response to the service/user that requested it previously. In this scheme, all interactions are carried out through the central service. In a choreography, there is no such central service, instead each participant acts on its own. Services are subscribed to events supplied by others in order to coordinate their activities and communicate with each other. The orchestration scheme is an example of a synchronous request/response communication and choreography of an asynchronous messaging communication scheme [17] [6, p. 203] [4].

Microservices expose functions that can be called over the network. This functions are called APIs. So developers can programmatically build applications calling

different functions from different microservices. Calls occur between pieces of code (i.e.,microservices) so this is intended to be machine-to-machine communication. The most common approaches to implement microservices are the SOAP standard and REST architectural style. Another aspect to consider when implementing microservices is the use of containers. This is helpful as individual services can be hosted in individual containers. Those containers enclose the microservices themselves, including all required libraries and data, which allows microservices to be self-contained.

2.4 Cellular Automata

"Cellular automata (CA) offer a powerful modeling approach for complex systems in which global behavior arises from the collective effect of many locally interacting, simple components" [19]. Many cellular automata application models have been published to study different phenomena, such as, the spread of forest fires [14], urban growth [10], snow crystal growth [15], road traffic [11], among others. Systems are comprised of many discrete elements with local interactions that are often modeled as cellular automata introducing finite differences and discrete variables in the system's differential equations [26, p. 6]. The resulting equations allow the system to be described in terms of a cellular automata properties: cellular space, set of states, local interactions, transition function, and boundary conditions. The cellular space is a grid of cells arranged in one, two or three dimensions. This is the place where a discrete representation of the system takes place. Each cell acting as a system constituent, changes its state applying a *transition function* which depends on its nearest neighbors including itself. The value returned by the transition function depends on the *set of states* that each system constituent can accept. Then, the next state of the whole system is determined after having applied this function over its constituents. Finally, *boundary conditions* are considered for one of two reasons. On the one hand, it is not possible to compute an infinite system. On the other hand, the system itself could have natural boundaries [25].

3 Related Work

Several CA programming environments have been implemented for scientific and educational purposes. These environments provide basic functionalities such as a modeling language/tool and a mechanism to build a visual representation of the CA during the simulation. In addition, some provide parallel processing support to fully fit the needs of scientific or real-world applications. Talia and Naumov [20] made a comparative survey of existing CA modeling environments according to a set of requirements [20, p. 360-361] such as support for distributed computing, extensibility and support for reproducibility, among others. These requirements define a well-suited CA environment. We are mainly concerned with those programming environments that provide extensibility and parallel processing support. In particular, they refer to projects such as CAM, CAME&L, JCASim and ParCeL-6.

CAM (Cellular Automata Machine) [12] is an accelerated device[5] developed specifically for high speed cellular automata processing. CA models are designed using the Forth programming language. It provides extensibility at hardware level, adding more "chunks" of hardware to the device.

CAME&L (Cellular Automata Modeling Environment and Library) comprise a set of C++ classes structured into components. These abstractions (classes) help users to establish the desired CA model. New components can be developed to fulfill the target problem requirements [20, p. 374-376]. In addition, it supports "parallel and distribute computing" on Windows platforms.

JCasin is also a general purpose system for CA modeling in Java. Parallel and distribute processing is supported via the CAComb Java package [20, p. 363-364]. Besides having the desktop application, it has a web application based on Java applets.

ParCel-6 is a computing library implemented in C for multiprocessor computer, clusters and grids. It can be linked with C and C++ programs. The software is also able to perform the automatic parallelization of the source code for multiprocessor machines [20, p. 364].

These tools show interesting features, such as; CAM specialized hardware, the automatic code parallelization and distribute/parallel processing capability in ParCel-6, and the web interface proposed by JCasim. However, each novel functionality is tied to a specific tool, i.e; much of the efforts in this field have been carried out in order to develop new tools (with new functionalities), but not in order to integrate existing or new ones. In their work "Simulation Engine for Cellular Automata Models" [1] the authors consider this problem suggesting a Microkernel Architecture (a.k.a plug-in[6] architecture). This approach provides extensibility combining existing tools to deal with modeling and high performance simulation of CAs. Unfortunately, a concrete implementation of such environment is not available at this time. In this work, we present a Microservice Architectural Style with its corresponding software prototype focusing in the extensibility requirement.

4 The Parallel Programming Environment

We developed a prototype of a parallel programming environment for general purpose cellular automata. This prototype exhibits a microservice architectural style, as shown in Fig.1, supported with containerization technologies (e.g., Docker and Docker Compose). This approach allows the resulting application to be; accessible, portable, extensible, easy to develop, deploy and maintain. We adopt the REST architectural style for service communication. The proposed prototype provides a code editor that can be put into operation through the web. This web code editor was inspired by Parallware Trainer[7] [3] design. Parallware Trainer provides code edition,

[5] CAM-8 the most recent version of this project uses a Sun workstation, which runs the STEP control software, as a host.

[6] Plug-in is a software component that adds a specific feature to a computer program.

[7] A general purpose parallel programming environment for C and FORTRAN.

analysis and parallelization with compiler directives (OpenMP and OpenACC) in a monolithic fashion. Unlike Parallware Trainer, we supply these functionalities as microservices. Each microservice represents an independent developed functionality attached to a central service. In particular, attached microservices support the parallel programming process in CA models.

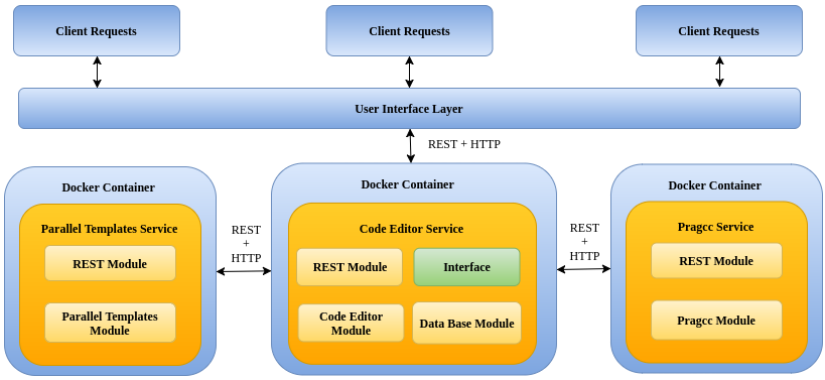


Fig. 1. Cellular automata parallel programming environment microservice architecture

This section describes the approach we adopted to turn a given parallel programming tool into a service supporting CA modeling. Additionally, we present architecture and implementation details for the prototype.

4.1 Turning parallel programming tools into services

The simplest method to build a parallel program nowadays, is to build a sequential version of it, then apply parallelism incrementally until the desired parallelized version is achieved. However, the ease in which parallelism can be applied to a sequential program depends clearly on the way in which it is structured, i.e., how sections of code, data structures and functions are organized. To guarantee the easy parallelization of a cellular automata model, we assume that cellular automata models must start from a sequential template that facilitates parallelism. Templates that facilitate parallelism are called parallel programming patterns. In particular, the kind of computation that is carried out in CA models, is called *stencil* in parallel programming patterns terminology.

We provide the user with a stencil template (a CA model in C99 programming language) by means of the **Parallel Templates (PT)** service. This template is then completed by the user through the **Code Editor (CE)** service and parallelized with the **Pragcc (PC)** service. The whole application or proposed prototype is then called **Coder**. Services details are described below as well as the operation of the complete application.

4.1.1 Parallel Templates (PT)

[22] is one of the developed tools which ensure a CA model follows the *stencil* [13, p. 89-90] parallelism pattern. PT takes the properties of the desired cellular automata model in YAML format and returns a C99 source code with such properties

set. This code must be completed by the user to fit the needs of the desired CA model. Some models only require the initialization and system dynamics functions to be completed whilst others exhibiting complex behaviors requires the edition of other template functions. The resulting template deals with function calls and simulation details.

A REST module was developed to expose PT as a service. This module receives a request from the code editor service, as shown in Fig.1. The body of the request contains a JSON object where a YAML document resides. This document is extracted and passed to the PT module. The PT module extracts the CA properties defined in the document and returns a C99 source code and parallelization metadata[8]. Finally, the resulting code is packaged into a JSON object by the REST module and sent back through the network to the CE service.

4.1.2 Code Editor (CE)

[21] C99 code edition is performed by the user through a web interface. Changes are maintained by the CE service. In the proposed architecture, Fig 1, CE fulfills the role of central service, i.e., this service is in charge of dealing with user requests and orchestrating other services according to the requested functionality (e.g., CA model templates or parallelization). Developed services must communicate with CE in order to supply to it with additional functionalities.

4.1.3 Pragcc

[23] Code parallelization functionality is provided by the Pragcc service. When code parallelization is required from the user interface the CE service sends the latest version of C99 source code and parallelization metadata in a request to the Pragcc service. Pragcc REST module manages the request, extracts the C99 source code and parallelization metadata, from the body of the request and sends them to the Pragcc module. This module then performs C99 code parallelization driven by the parallelization metadata. Finally, it returns a C99 source code annotated with compiler directives (e.g.,OpenMP or OpenACC).

4.1.4 Coder

A cellular automata project is generated with the assistance of Coder, as shown in Fig 2. It provides a user interface to specify the properties of the desired CA model (3). Properties such as, the size of the cellular space (Matrix number of rows and cols), type of values that will be held in the cellular space (States Type), number of generations the system is intended to evolve (Number of Generations) and how the parts of the system will interact (Neighbor name) are required. In addition, the project requires a description, name and the selection of a base template (2). Created projects are listed in the projects manager section (1).

A project is opened as shown in Fig 3. The code edition interface includes; the files viewer (1), download project option (2), sequential code editor (3), parallelized

[8] This metadata is intended to be used by the parallelization service to drive the C99 source code parallelization.

code viewer (4), options (5), and a console to show the user the results of each option (6).

A typical cellular automata project consists of three files: *stencil.c*, *parallel.yml*, and a *Makefile*. The first one is the base code of a cellular automata model in C99 programming language. The second is the parallelization metadata required to parallelize the cellular automata source code. The last one is a file created to assist the user when running the C99 application.

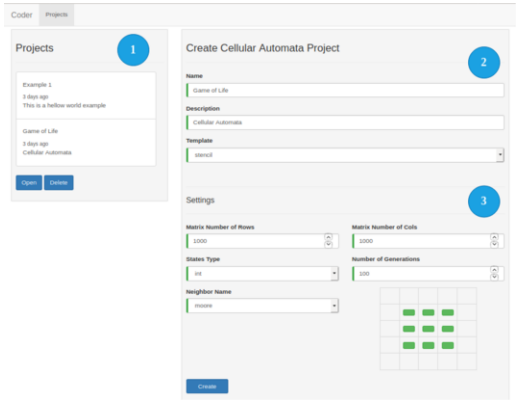


Fig. 2. Coder: Project creation interface.

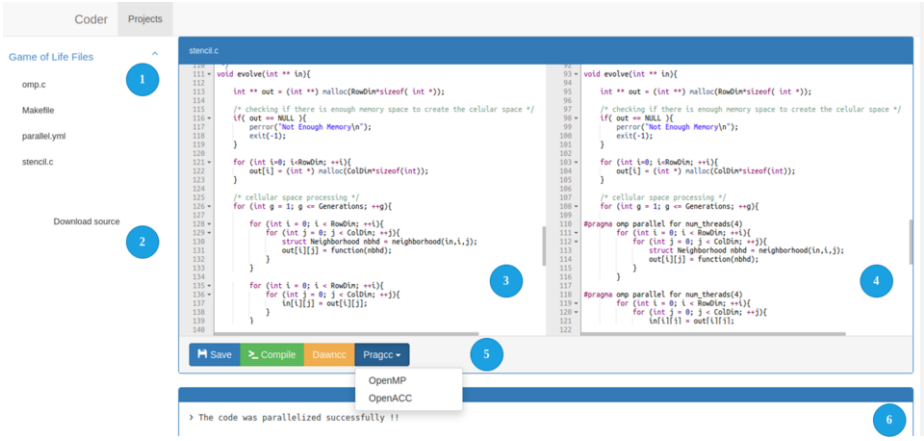


Fig. 3. Coder: Code edition interface.

5 Use Case

Cellular automata are a vehicle to explore a variety of complex behaviors emerging from the interaction among the parts that make up dynamical systems. One of the most studied CA model is the Conway's Game of Life.

The Game of Life is a two-dimensional CA designed by the British mathematician John Horton Conway in 1970. It is used to simulate life, death and survival, through the evolution of an initial pattern of "living" cells. We can reach one of the following states after having evolved the system for several generations: extinction,

where there are no "living" cells in the cellular space; stability, all "living" cells remain in the same state; oscillation, "living" cells enter into an endless cycle [7, p. 32].

The Game of Life CA is rich in patterns and complexity appearing from a simple system dynamics function. The deterministic nature of this function is convenient to test new parallel programming and simulation environments for CA models, since given an initial configuration at time t we can determine in advance which pattern is going to be generated at time $t+1$ for testing purposes. This is important when we talk about parallelizing an application, because the parallelization process must be guided by validations of results every time we perform any code optimization. Finally, what concerns us in this section, is to check through an experiment that the CA model generated using Coder is correct.

5.1 Setting up the Game of Life CA model in Coder

A cellular automata project was generated with the assistance of Coder's project creation interface. The initial parameters provided by this interface for the experiment are shown in Fig 2. Subsequently, the *stencil.c* template was edited with the assistance of the code edition interface. This file provides the user with five functions which make up the CA simulation model: *initialize*, *function*, *neighborhood*, *evolve*, and *main*. In the *initialize* function, the user must give an initial configuration for the cellular space. The initial configuration considered, is shown in Fig 4b, "living" cells are represented with ones (black color), "dead" cells with zeros (white color).

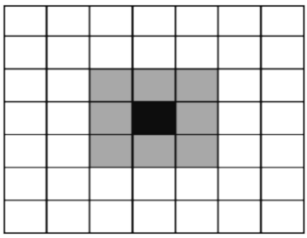
After evolving the system for 100 generations and carrying out code optimizations (parallelization), we expect each experiment to start in configuration Fig 4b and end in the configuration shown in Fig 4c.

The set of rules which determine the dynamics of the Game of Life system were described in *function*. The function implementation considers the following:

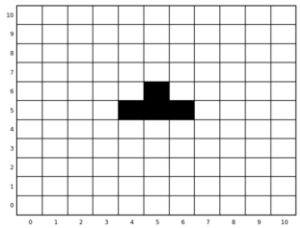
- Death, a "living" cell dies if it has less than two or more than three neighbors alive.
- Birth, a "dead" cell comes to life if it has exactly three living neighbors.
- Survival, a "living" cell stays alive if it has exactly two or three living neighbors.
- Remains lifeless, a "dead" cell remains in the same state if it has less than three or more than three non "living" neighbors.

Rules described above are considered under the Moore neighborhood, shown in Fig.4a, with periodic boundary conditions. The neighborhood [9] or area that will be considered as the neighborhood for every cell in the cellular space must be implemented in the *neighborhood* function. Finally, the implementation of *evolve* and *main* functions are provided by the template.

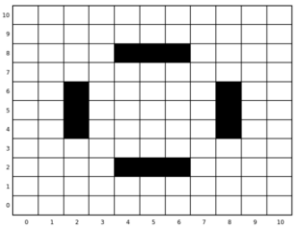
[9] In other literatures the neighborhood pattern is called the *stencil*.



(a) Moore neighborhood



(b) $t = 0$



(c) $t = 100$

Fig. 4. Cellular Automana System Setting

5.2 Experimental Setup

The experiment considers different sizes for the cellular space: 250^2, 500^2, 750^2, 1000^2, 1250^2, 1500^2. The above values serve to observe the quality of the optimizations suggested by the tool as the size of the cellular space increases. For each grid size, 100 generations of the CA are simulated. In order to reduce the number of outliers (generations which took above the average time to calculate), the processing time for each size of the cellular space is calculated 10 times. The total time to calculate 100 generations is measured in seconds. The hardware and software used in the experiment are described below.

- CPU
  - · Model name: Intel(R) Xeon(R) CPU E5-2609 v3
  - · Clock Speed: 1.90 GHz.
  - · Thread(s) per core: 1
  - · Core(s) per socket: 6
  - · Socket(s): 2
- GPU
  - · Model name: Nvidia Quadro K620
  - · Clock Speed: 1124 MHz
  - · Number of Multiprocessors: 3
  - · CUDA Cores: 384 (128 per SM).
  - · Bandwidth: 29 GBps.
- Software
  - · pgc++ compiler in a Docker Container.

5.3 Analysis and Results

The Game of Life CA model was developed and optimized with the assistance of the proposed prototype. Optimizations were performed with OpenMP and OpenACC compiler directives. These were used to carry out parallelism in multi- and many-cores environments respectively. The execution time of the model was recorded considering three programming approaches: sequential, multi- and many- core. The time taken to calculate a number of generations for different grid sizes was measured and analyzed for each approach. Results are presented in Fig 5a and Fig 5b.

In Figure 5a, when the first set of values were executed, for the grid size, between

500 and 1500, it is not clear which OpenMP or OpenACC approach dominated (had less processing time). However, these two behaved better than the sequential approach as was expected.

Additional information regarding which approach behaved better between 500 and 1500 grid sizes, can be obtained by calculating the *speedup*. The speedup shows the effect on performance of an application after any resource enhancement.

In general, speedup is defined as follows:

$$S = \frac{T_f}{T_e} \tag{1}$$

Where T_f is the processing time in seconds of a primary programming approach. T_e is the processing time in seconds of an enhancement performed over the primary approach. In consequence, $S = 0$ implies that the new solution makes the previous one worse; $S = 1$, implies no significant improvement; $S > 1$, there was improvement of the enhanced approximation with respect to the previous one.

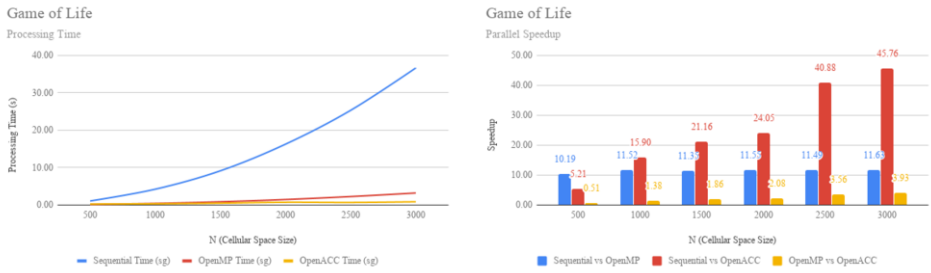
The speedup for each grid size considered for the cellular space in the Game of Life experiment is depicted in Fig 5b. For grid size 500^2 the OpenMP approach runs about 10 times faster than the sequential approach whilst the OpenACC runs just 5 times. The OpenACC behavior may be due to the fact that the amount of data to be processed is relatively small. Hence the amount of parallelism immersed does not justify the activities the GPU must carry out to perform the computation. Applications that access parallel processing must incur an additional computational cost because multiple execution units which are kept active concurrently. On top of that, some GPU enabled computers must incur the data transfer cost.

By comparing the OpenMP vs OpenACC approaches, Figure 5b (in yellow color), we can obtain additional information about which is the best alternative for the 500^2 grid size. The value that is observed in Fig 5b tells us that the first approximation is better than the second given that the value of the speedup is rounded to zero. In other words, the OpenACC approach is not an improvement of the OpenMP approach for this specific grid size.

In general, the speedup of the OpenACC approach, with respect to the sequential increases (Sequential vs OpenACC), as there is more data to be processed (data parallelism). On the other hand, the OpenMP approach with respect to the sequential (Sequential vs OpenMP) seems to stay constant, i.e., there is no significant improvement as the amount of data to be processed increases.

6 Microservice Approach Test

An online programming environment proposes a big challenge as users are constantly interacting with the application, using different services to approach the parallel programming needs. Software development comprises a variety of methods to ensure an application is able to meet the growing user's demand without affecting the quality of service. For the purpose of this work, we summarize the concept of quality of a service as the time it takes to serve a user request. As the quality of service can be affected by aspects such as the speed of Internet connection and the complexity



(a) Moore neighborhood (b) $t = 0$

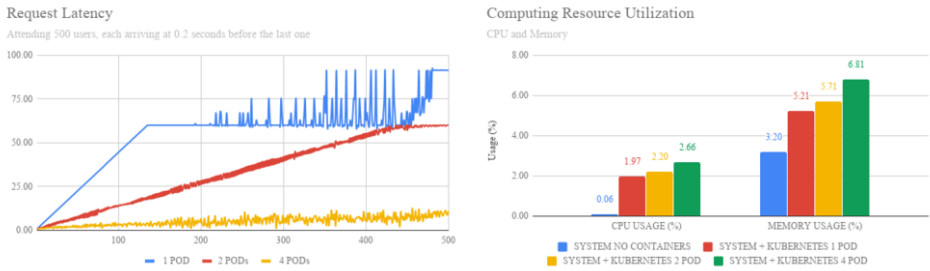
Fig. 5. Cellular Automana System Setting

of the task to be performed by the service, we consider a local area network and the Pragcc as the microservice to be evaluated on this experiment. Aspects such as assessment of vulnerabilities in information security and other metrics that measure the quality of service (for example throughput and availability) are not considered.

6.1 Experimental Setup

An stress test was performed to evaluate the behavior of the quality of the service as the user demand increase. The test includes the attention of 500 users in total. Each user requests the Pragcc service to parallelize the Game of Life source code, showed the in the previous section. New users (requests) arrives at 0.2 seconds after the previous one. The time to serve users request is measured, but also the impact in CPU and Memory of the deployed services in steady state and during the stress test. The Pragcc service is scaled up using Kubernetes orchestration services. Settings with 1, 2 and 4 Kubernetes Pods were considered. Pods are scheduled in a single machine in a bare-metal environment. The hardware and software used in the experiment are described below.

- CPU
  - Model name: Intel(R) Core(TM) i7-4790
  - Clock Speed: 3.60GHz
  - Thread(s) per core: 2
  - Core(s) per socket: 4
  - Socket(s): 1
- Software
  - Ubuntu 18.04 Operative system.
  - Docker, the software used to deploy containers. Containers in turn package the deployed microservices.
  - Kubernetes, the containers orchestration software includes scheduling, distributing containers across a cluster of servers and scaling individual or a set of containers.
  - JMeter, a load testing used to measure an analyses the performance of a variety of services.



(a) Requests Latency (b) Kubernetes Pods CPU and Memory Usage

Fig. 6. Cellular Automana System Setting

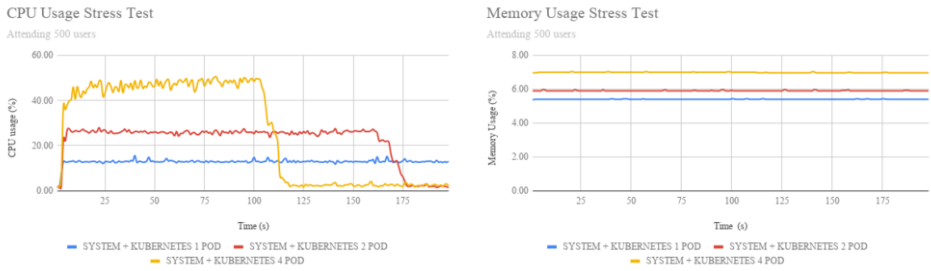
6.2 Analysis and Results

500 requests were sent to the Pragcc service. Each request sent 0.2 seconds after the previous one. Since the time between requests is short, it is likely that certain number of requests will accumulate on the system. The Fig 6a shows this behavior for 1, 2 and 4 Kubernetes Pods. That is, for 1, 2 and 4 replicas of the Pragcc service meeting the same user demand. Fig 6a depicts the increase in latency for users requests as the number of concurrent requests in the system increase. Successful requests will receive a 200 status code from the server. Otherwise, failed requests will get a 503 error code. As services are behind a proxy server (Nginx), by default it sends a 503 error code for an user request when it takes more than 1 minute to be answered. The blue line describes latency when a single Pod (one instance of the Pragcc service) is dealing with the user demand. The breaking point in the graphs at user request 136 depicts that the limit of 1 minute of latency per user request was reached. From this point, subsequent requests will fail (503 error code) dramatically. The fluctuation behavior described before user request number 210 is the time taken by the server to tell the user that his/her request was failed (503 error code).

Scaling the number of Kubernetes Pods available in the system over the round robing load balancing Kubernetes's scheme, has the effect of increasing the number of users the system is able to attend. Surely, interactive applications requires response times lower than 1 minute. Finally, 4 Pods were required to reduce latency to an average of 5.33 seconds per request. Other services may require a different configuration to meet the user needs.

The impact of scaling the application was measured in terms of CPU and Memory usage, see Fig 6b. In terms of CPU, deploying a single Kubernetes Pod implies a 2% increase in the use of processing resources. Then, the percentage of CPU usage for subsequent Pod deployments increases linearly in multiples of 0.23%. The same lineal behavior is faced for Memory usage.

For stress testing, the use of CPU and Memory resources are shown in Fig 7a and Fig 6b. keeping in mind that the test consists of serving 500 users, each user making a request to the system 0.2 seconds after the previous one. Then, the last request will be sent to the application in the second 100 ($500 * 0.2$). The above may be another indicator that allows us to determine which configuration is best suited



(a) CPU Usage (b) Memory Usage

Fig. 7. Stress Test

given the demand of users under this particular scenario. We can determine that desirable scenarios are those whose relaxation time is before or near 100 seconds, as shown in Fig 7a, yellow line. The other red and blue line scenarios are those where failed requests were obtained.

Another measurement that can be obtained from this result is the amount of Pods that the computer can support under the described user demand. Since for 4 Pods, the computer reaches values close to 50% for resource utilization. 8 Pods are expected to reach 100% utilization at a given time during a stress test. In addition, for a stress test with 8 pods is expected to reach the relaxation in a time lower than 100 seconds. On the other side, the Memory usage remains constant during the stress test, see Fig 7b. It also does not show significant differences compared to Memory usage when the stress test is not running Fig 6b.

7 Conclusion

A parallel programming skeleton tool and a parallelizing compiler were integrated to assist the development and parallelization of a General Purpose cellular Automata. Performance tests were carried out to ensure the tools is working properly. As the performance of the generated code depends on the tools integrated into the system. It is required to integrate tools with more elaborate code analysis.

Considering the microservice-based approach, a specific test was conducted to study the behavior of the system under stress. From the results obtained we could derive two indicators that allow us to determine for this scenario in particular what is the best configuration in terms of Kubernetes Pods. The first indicator is derived form the behavior depicted in Fig 6a. This indicator establish 4 Pods to be the desirable setup to meet the users demand among the 3 Pods setups considered. The last indicator is derived from Fig 7a. And establish 4 Pods to be the desirable setup. As this setup ensures that the relaxation of the system arrives in time less or close to the time on which the last user request arrives to the system. Finally, Fig 6a and 7a are closely related, i.e., when the relaxation time of a given Kubernetes Pods setup is grater than the time on which the last user request is send to the system, that setup will present failed requests.

In conclusion, several attempts to parallelize CAs end in desktop applications.

That is the case for CAM, CAME&L, JCASim and ParCeL-6. These tools do not provide portable parallelization methods and mechanisms. On top of that, the learning curve they pose for scientists is relatively large. Moreover, they are not easy to improve. The parallel programming environment we propose for general purpose CAs parallelization was developed to be improved because of its microservice approach. The above implies that each optimization or code enhancement can be performed over CA models following a microservice approach. New microservices can then be integrated into prototype easily.

Future work could provide modifications regarding the prototype architecture. This is because the communication pattern present in the application adds more complexity to the **Code Editor** service. Mark Richards on [16, p. 31] proposes a better approach to deal with this complexity by attaching all microservices to the network interface. Services will then be seen as disjointed utilities providing services to the interface. This modification will reduce the complexity of the **Code Editor** service but also destroy the unnecessary dependence on which all operations (requests to other services) must be carried out through it, as shown in Figure 1.

CAM, CAME&L, JCASim and ParCeL-6 are specific purpose tools. The proposed prototype can be generalized to deal with different problems and different programming models as more tools are integrated.

References

[1] Marcelo O Alaniz, Fabricio Bustos, Verónica Gil-Costa, and A Marcela Printista. Motor de simulacion para modelos de automata celular. In *2nd International Symposium on Innovation and Technology ISIT*, pages 28–30, 2011.

[2] Appentra. Products — appentra. https://www.appentra.com/products/. (Accessed on 07/31/2019).

[3] Manuel Arenaz, Sergio Ortega, Ernesto Guerrero, and Fernanda Foertter. Parallware trainer: Interactive tool for experiential learning of parallel programming using openmp and openacc. In *EduHPC-17: Workshop on Education for High-Performance Computing.*, 2018.

[4] Björn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. In *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*, pages 1–6. IEEE, 2016.

[5] Camunda. Microservices and bpm whitepaper — camunda bpm. https://camunda.com/learn/whitepapers/microservices-and-bpm/?utm_source=stack&utm_medium=cpc&utm_campaign=newchannel_test_march. (Accessed on 04/02/2018).

[6] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[7] Ryno Fourie. *A parallel cellular automaton simulation framework using CUDA*. PhD thesis, Stellenbosch: Stellenbosch University, 2015.

[8] Inside HPC. Shifter - docker containers for hpc - insidehpc. https://insidehpc.com/2018/04/shifter-docker-containers-hpc/. (Accessed on 04/16/2018).

[9] Ann Mary Joy. Performance comparison between linux containers and virtual machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, pages 342–346. IEEE, 2015.

[10] Uttam Kumar, Chiranjit Mukhopadhyay, and TV Ramachandra. Cellular automata calibration model to capture urban growth. *Boletin Geologico y Minero*, pages 285–99, 2004.

[11] Sven Maerivoet and Bart De Moor. Cellular automata models of road traffic. *Physics Reports*, 419(1):1–64, 2005.

[12] Norman Margolus. Cam-8: a computer architecture based on cellular automata. *arXiv preprint comp-gas/9509001*, 1995.

[13] Michael D McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[14] Sang Il Pak and Tomohisa Hayakawa. Forest fire modeling using cellular automata and percolation threshold analysis. In *American Control Conference (ACC), 2011*, pages 293–298. IEEE, 2011.

[15] Clifford A Reiter. A local cellular model for snow crystal growth. *Chaos, Solitons & Fractals*, 23(4):1111–1119, 2005.

[16] Mark Richards. *Software architecture patterns*. O'Reilly Media, Incorporated, 2015.

[17] Chris Richardson. Building microservices: Inter-process communication. https://www.nginx.com/blog/building-microservices-inter-process-communication/. (Accessed on 04/18/2018).

[18] S Senthil Kumaran. *Practical LXC and LXD*. Apress, Berkeley, CA, 2017.

[19] Domenico Talia. Cellular processing tools for high-performance simulation. *Computer*, 33(9):44–52, 2000.

[20] Domenico Talia and Lev Naumov. Parallel cellular programming for emergent computation. In *Simulating Complex Systems by Cellular Automata*, pages 357–384. Springer, 2010.

[21] Argüelles A Vivas A and Sanabria J. Donaurelio/coder: the base for a web-based parallel programming environment build over a microservice approach. https://github.com/DonAurelio/coder. (Accessed on 04/30/2018).

[22] Argüelles A Vivas A and Sanabria J. Donaurelio/parallel-templates: C99 templates for parallel programming. https://github.com/DonAurelio/parallel-templates. (Accessed on 04/30/2018).

[23] Argüelles A Vivas A and Sanabria J. Donaurelio/pragcc: A source to source parallelizing compiler initiative. https://github.com/DonAurelio/pragcc. (Accessed on 04/30/2018).

[24] Deepak Vohra. *Kubernetes microservices with Docker*. Apress, 2016.

[25] Weimar. 2.3 boundary conditions. http://www.jcasim.de/main/node6.html. (Accessed on 10/29/2017).

[26] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of modern physics*, 55(3):601, 1983.