Artificial Embryogeny and Grid Computing

Sylvain Cussat-Blanc, Hervé Luga, Yves Duthen

Institut de Recherche en Informatique de Toulouse
Université de Toulouse – CNRS - UMR 5505
2 rue du Doyen-Gabriel-Marty
31042 Toulouse Cedex 9, France

Technical Report

{cussat, luga, duthen}@irit.fr

http://www.irit.fr/∼Sylvain.Cussat-Blanc

April 10, 2008

Abstract

In Artificial Life, the production of new artificial creatures always needs more and more computation power. Whereas artificial morphogenesis methods construct complete creatures using blocks, artificial embryogeny develops smaller creatures starting from a unique cell. To obtain a complete creature, organized in tissues and organs, we propose a developmental model in which cells are coded as threads. This massive parallel architecture allows the simulation of an organism development on multi-core or multi-processor machines. In most cases, evolutionary algorithms and especially genetic algorithms are used to create our creatures. Their algorithms take a lot of computation time to find an environment-adapted creature. In order to reduce the computation time, genetic algorithms have already been parallelized, but, in most cases, using a supercomputer. This solution is very expensive and not easily scalable. In this report, we first present our model of artificial embryogeny, *Cell2Organ*. Then, we propose an implementation of genetic algorithms for artificial embryogeny using a computational grid and ProActive middleware.

Contents

Chapter 1

Introduction

Several models exist for creating artificial creatures. These models use different levels of abstraction to produce creatures of various shapes and sizes. Whereas the morphological approach produces relatively large creatures as in [Sims(1994), Fukunaga et al.(1994), Lassabe et al.(2007)], embryogenic models produce creatures composed of hundreds of cells starting from a unique cell [Chavoya & Duthen(2007), Dellaert & Beer(1994), Stewart et al.(2005)].

This report details our model of cellular development [Cussat-Blanc et al.(2007a), Cussat-Blanc et al.(2007b)], *Cell2Organ*. For the purpose of creating complete creatures composed of different organs, we propose a model able to produce organisms that perform specific functions. These organisms respect the biological definition of an organ. In other words, they are a "specialized cell regrouping that performs specific function or a group of functions". Our model contains an environment with a simple artificial chemistry [Rasmussen et al.(2003), Dittrich et al.(2001), Hutton(2007), Ono & Ikegami(1999)] and cells that perform different actions. Cells are able to self-replicating and to specialize themselves to optimize specific actions instead of others. Moreover, we show that *Cell2Organ* can also produce simple creature shapes. The final aim of our project is to develop a complete creature starting from a unique cell.

Genetic algorithms are very demanding in terms of computing time and they need days to complete or even fail due to memory restrictions when the population size is large. It is particularly the case for artificial life where each evaluation can take more than one minute to develop an artificial creature, plant or organism. Indeed, creatures are developed in physical and chemical simulators that requiere important computation resources. Therefore, in order to create more and more realistic creatures, we need more and more computation resources. Two possibilities exist to increase them: supercomputers or computational grids. Because of the

price and the low possibility of evolution of the first architecture, we decide to use a computational grid. In this work, we use the French experimental grid, Grid5000[1].

This report is organized in two main parts. Chapter 2 presents our model of cellular development, *Cell2Organ*, starting with a review of existing model. Next, we give a description of the environment functioning and the mechanisms used by our artificial cell to interact with the environment. The possibilities of the model are then shown by the development of two types of organisms : a primitive organ able to move substrate in the environment and two creatures with particular morphologies. These experiments point to the possibility of simulating, in a simplified way, different approaches to organism growth. Chapter 3 presents our method to parallelize genetic algorithm using a computation grid. It first details 3 methods to parallelize genetic algorithm on supercomputer and also show that it is possible to apply it to computational grid. We next present ProActive[2], a grid programming middleware that provides an infrastructure abstraction of the grid. Finally, we present different experiments and theirs results of our genetic algorithm parallel version using ProActive and the french computational grid, Grid5000. This report concludes on a discussion and possible future works.

[1] www.grid5000.fr
[2] proactive.inria.fr

Chapter 2

Artificial Embryogeny : From Cell to Organ

This chapter presents our model of artificial embryogeny. It is organized in three sections. Section 1 presents related works about artificial creatures development, presenting artificial morphogenesis, cellular automata and already existing works about artificial embryogenesis. Section 2 presents our model of cellular development, *Cell2Organ*, starting with a description of the environment functioning and the mechanisms used by our artificial cell to interact with the environment. Section 3 presents different experiments using this model. The possibilities of the model are shown by the development of two types of organisms : a primitive organ able to move substrate in the environment and two creatures with particular morphologies. These experiments point to the possibility of simulating, in a simplified way, different approaches to organism growth.

2.1 Related works

2.1.1 Artificial morphogenesis

Several projects have tried to generate artificial creatures well adapted to their environment. For example, in his famous works, Karl Sims [Sims(1994)] uses blocks with different properties such as size, shape, contact sensor positions or block layout. Komosinski also creates Framsticks creatures [Komosinski & Ulatowski(1999)] using an equivalent architecture: blocks are replaced by sticks but creature functioning is comparable to Karl Sims' work: he uses a neural network to coordinate creature movements. Sims' work was improved by Nicolas Lassabe by using a more complex environment [Lassabe et al.(2007)]. Lassabe's creatures are able to climb a stairway or to practice skateboarding.

The aforementioned creatures use high level components to create their morphology and their behavioral controller. A more biological-inspired approach was introduced by Dawkins in [Dawkins(1986)]. Using simple rules to draw continuous segments, he developed a model able to create small graphic creatures. The addition of behaviors in these simple life forms allows the creation of a complex 2-D virtual world [Ventrella(1998a), Ventrella(1998b)] where small filiform creatures co-evolve in an environment composed of energy sources. Each creature has a vital energy level and must survive in the environment, looking for food produced by the death of other creatures. This model produces a complete ecosystem with its own food chain. Creatures are also able to reproduce among themselves to create new life forms. EvolGL [Garcia Carbajal et al.(2004)] is another 3D pond life project where creatures have different classes, such as herbivorous, carnivorous or omnivorous, which allows the emergence of survival strategies.

Using lower level components, cellular automata use neighborhood rules to evolve a cell matrix. The rules give the $t+1$ state of each cell according to the cell neighbor's t state. Using this method, John H. Conway [Gardner(1970)] creates interesting patterns such as gliders, pulsars, spaceships, etc.

2.1.2 Artificial Embryogenesis

One of the first works on artificial embryogenesis was that of Hugo de Garis [de Garis(1999)]. Using a cellular automata, he developed 2D shapes. The cellular automata rules were evolved with a genetic algorithm. The aim was to generate desired shapes like letters.

Another important goal of artificial embryogenesis is cell specialization. Different works on cell specialization already exist. In most cases, they use a Genetic Regulatory Network (GRN), just as in nature.

In nature, the cells of an organism can have different functions, all of which are specified in the organism's genome and regulated by a Gene Regulatory Network (GRN) [Davidson(2006)]. Cells get input signals from the environment thanks to receptor proteins. The GRN, described in the organism's genome, uses these signals to activate or inhibit the transcription of different genes in the messenger RNA, the future cell's DNA protein template. The expression of these genes will specify the cell's functions. Figure 2.1 shows (in a very simplified way) the functioning of the GRN.

This nature inspired model was designed by Banzhaf in [Banzhaf(2003)]. In this work, each gene beginning is marked by a starting pattern, named "promoter". Before the coding of the gene itself, enhancer and inhibitor sites allow the regulation

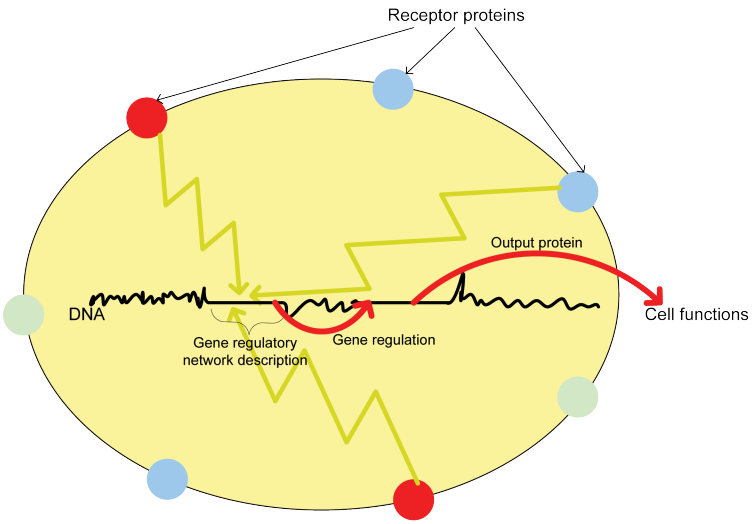

Figure 2.1: Scheme of the GRN action in cell duplication.

of its behavior. In [Chavoya & Duthen(2007)], Chavoya and Duthen introduced another model in which the gene regulation system is encoded at the beginning of the genome. It consists of a series of inhibitor sites, enhancer sites and regulatory proteins. The production of each regulatory protein is conditioned by the inhibitor/enhancer sites. The concentration of this protein determines the cell function's activation or inhibition : if the concentration level is over a certain threshold, the gene is activated and so are the corresponding functions.

A different approach is the Random Boolean Network (RBN) first presented by Kauffman [Kauffman(1969)] and reused by Dellaert [Dellaert & Beer(1994)]. A RBN is a network where each node has a boolean state: activate or inactivate. The nodes are interconnected by boolean functions, represented by edges in the net. The state of a node at time $t + 1$ depends on its particular boolean function applied to the values of its inputs at time t. The mapping to the gene regulatory network is simple: each node of the net corresponds to a gene and each boolean function represents the activity regulation of the gene. The cell function will be determined during the interpretation of the genome.

Eggenberger Hotz [Eggenberger Hotz(2004)] imagines a concept able to produce a simple creature with a user defined shape able to move in an environment just using a gene reguation network. Cells rhythmically emit molecules which modify the adhesion properties between cells and between cells and the environment. He develops a simple simulator and develop a T-shape that grows and move in the environment.

The aim of our work is to make a bridge between artificial morphogeny and artificial embryogenesis to produce virtual creatures. We decide to use the hypothesis

that blocks and sticks can be considered as organs, that is to say body parts of the creature able to carry one or more specific functions. Using developmental techniques of creature growth, we could create these organs starting from a single cell. In this way, the cell must be able to specialize itself into a cell more adapted to the environment. The cell organization in tissues (that is in cell groups that have the same function) and then the tissue organization will allow the creation of organs. After creating a library of organs, we will just have to assemble them to create a creature adapted to the environment with a morphological approch. This paper presents the embryogenic approch of the problem, and especialy the creature shape development. The next section details the model, starting with the environment and, then, showing the cell mechanisms.

2.2 *Cell2Organ* : a new cellular developmental model

2.2.1 The environment

To reduce the simulation computation time, we implement the environment as a 2-D toric grid. This choice allows an important decrease in the simulation's complexity.

The environment contains different substrates. They spread in the grid, minimizing the variation of substrate quantities between two neighbor crosses of the grid. This spreading is enacted in two stages, as illustrated by Figure 2.2

- First, the substrate spreads to the 4 cardinal points.
- Then, if the substrate quantity is sufficient, the substrate spreads to the diagonal crosses.

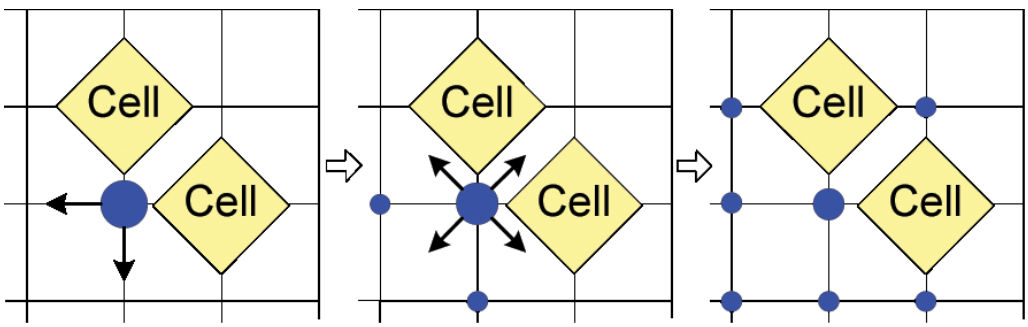

Figure 2.2: Example of spreading substrate in the environment.

Our model integrates a highly simplifed model of artificial chemistry. Many works exist on artificial chemistry [Dittrich et al.(2001), Rasmussen et al.(2003)]. In

these works, the artificial chemistry is highly developed and allows a good simulation of cell mechanisms. For example in [Ono & Ikegami(1999)], the cell division and the cell membrane formation and maintaining are highly realistic. However, the complexity of such a model is very great and does not support a high number of cells. In our model, the properties of artificial chemistry defined in [Dittrich et al.(2001)] have been simplified.

Our molecules, named substrates, have different properties like diffusion speed or color, and can interact with other substrates. This interaction between substrates can be viewed as a typical chemical reaction: using different substrates, the transformation will create new substrates, emitting or consuming energy. For example, the transformation $2A + B \rightarrow C$ (+50) denotes that, using 2 units of substrate A and 1 unit of B, a unit of C is created, emitting 50 units of energy. To reduce the complexity at the maximum, the environment contains a list of avalaible substrate transformations. The substrate reactions can only be triggered by cells. Then, in the previous example, from a biological point of view, C can viewed as waste from a cell which has the ability to convert A and B into energy.

To modify this environment, cells interact with the environment. They have different abilities and must perform a global action defined by the user. This action can be very diverse: harvest substrate, modify environment, create shapes or simply survive as long as possible. The next section describes cell functioning.

2.2.2 The cells

Cells evolve in the environment, more precisely on the environment diffusion grid. Each cell contains sensors and has different abilities (or actions). An action selection system allows the cell to select the best action to perform at any moment of the simulation. Finally, a representation of a GRN is inside the cell to allow specialization during duplication. Figure 2.3 is a global representation of our artificial cells.

Sensors

Each cell contains different density sensors positioned at each cell corner. Sensors allow the cell to measure the amounts of substrates available in the cell's Von Neumann neighborhood. For each substrate in the environment, a corresponding sensor exists. Only this corresponding sensor can compute the density of the substrate. The list of available sensors and their position in the cell is described in the genetic code.

For example, in Figure 2.3, the cell has sensors for B and D substrates in the

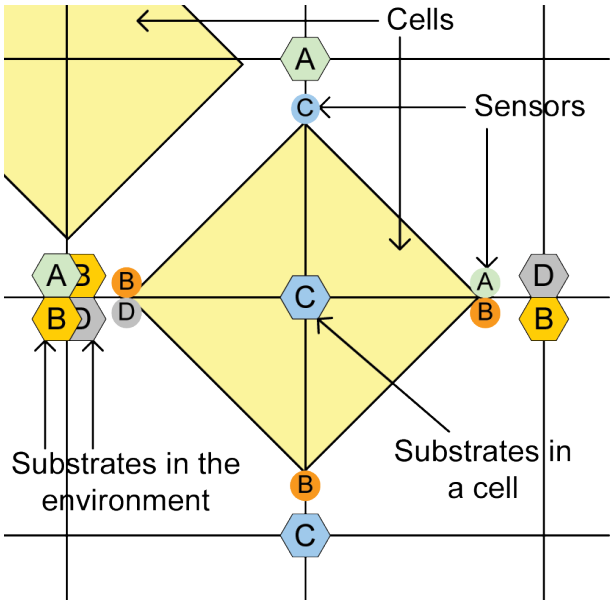

Figure 2.3: Scheme of a cell in an artificial environment. It contains substrates (hexagons) and corresponding sensors (circles)

left corner. The results of the measure of the corresponding substrate densities are :

- 2 units for B substrate because of the presence of 2 units of B substrates in the left cross of the cell,
- 1 unit for D substrate.

Actions

To interact with the environment, cells can perform different actions:

- The *substrate transformation* allows the cell to trigger a substrate reaction as previously described. To start, all the needed substrates on the left part of the equation must be present in the cell, that is, the needed substrates must be in the same intersection as the cell. In result of the reaction, the vital energy is increased or decreased (depending of the reaction properties), the needed substrates are destroyed and the new substrate is created.
- The cell can *absorb* or *reject* substrates in the environment. These two actions allow the cell to move substrates from one place to another. These actions, and particulary the first one, are important to trigger a substrate transformation.
- The *duplication* action allows the cell to create a new cell. We give details about this action in the next section.

- *Survive* is an action that allows the cell to wait for a signal from the environment to do something.

- *Apoptosis* allows the cell to autodestruct. This action can be useful to free a place for a more specialized cell for example.

The previous list is not final. Our model must be able to allow us to add new actions easily. Like sensors, all actions are not available for the cell: the genetic code will give the available action list.

Cells contain an action selection system. This system is inspired by classifier systems [Holland & Reitman(1978)]. It uses data given by sensors to select the best action to perform. The selection system can be viewed as a rule database, where each rule is composed of three parts:

- The *precondition* describes when the action can be triggered. It is composed of a list of sensor value intervals that describe the best substrate densities in the neighborhood to trigger the action.

- The *action* gives the action that must be performed if the corresponding precondition is respected.

- The *priority* that allows the selection of only one action if more than one can be performed. The higher the coefficient, the more probable is the selection of the rule.

Action selection rules can be, for example :

$$\begin{array}{lcl} (SensorA = 1) & and & (3 < SensorC < 7)\ and \\ (SensorB = 0) & \rightarrow & (ActionA)\ (23) \\ (SensorC = 3) & \rightarrow & (ActionB)\ (17) \\ & \rightarrow & (ActionC)\ (13) \end{array}$$

In this example, $ActionA$ will be performed if and only if $SensorA$ value is equal to 1 unit, $SensorB$ does not detect the presence of its associate substrate and $SensorC$ value is more than 3 units and less than 7. $ActionC$ does not contain a precondition. It means that this action can always be performed. The priority coefficients sort actions in the order $ActionA > ActionB > ActionC$ if different actions are possible.

In the list of possible actions, the cell can duplicate itself. We will now examine this action in detail.

2.2.3 Duplication

The duplication is an action that can be performed by the cell if and only if the next conditions are respected:

- The cell must have at least one free neighbor cross to create the new cell.
- The cell must have enough vital energy to perform the duplication. The vital energy level need is defined during the specification of the environment.
- A list of conditions can be added during the modelization of the environment. For example, some substrates can be needed to create a new cell.

The new cell created after duplication is completely independent and interacts with the environment. During duplication, the cell can be specialized to optimize a group of actions instead of others actions. In nature, this optimization is carried out by the GRN. In our model, we imagine a simplification of the GRN. Each action has an efficiency coefficient that corresponds to the action optimization level : the higher the coefficient, the lower the cost of vital energy. Moreover, if the coefficient is null, the action is not yet available for the cell. Finally, the sum of efficiency coefficients must remain constant during the simulation. In other words, if an action is optimized increasing its efficiency coefficient during a duplication, another efficiency coefficient (or a group of them) has to be decreased.

The cell is specialized by varying the efficiency coefficients during duplication. The rules of these variations are given by the GRN. We use a network to simulate the GRN:

- the network's nodes represent cell actions with their efficiency coefficients,
- the network's edges are weighted. The edge's weight (a real number in the interval [0,1]) represents the efficiency coefficient quantity that will be transferred during the duplication.

Figure 2.4 is an example of our GRN. $(A, 35\%)$, $(B, 25\%)$, $(C, 17\%)$, $(D, 23\%)$ are cell actions with their associated efficiency coefficient. The edge between 2 actions represents the amount of efficiency coefficient that will be transferred during duplication. For example, the weighted edge between A and B means that after one duplication, 30 percents of the A action efficiency coefficient will be transferred to the B action. After four duplications, we can see that the actions B and C respectively have been optimized to the detriment of the actions A and D. According to this simple example, we can say that the cell function of the organism has been specialized during the duplication process.

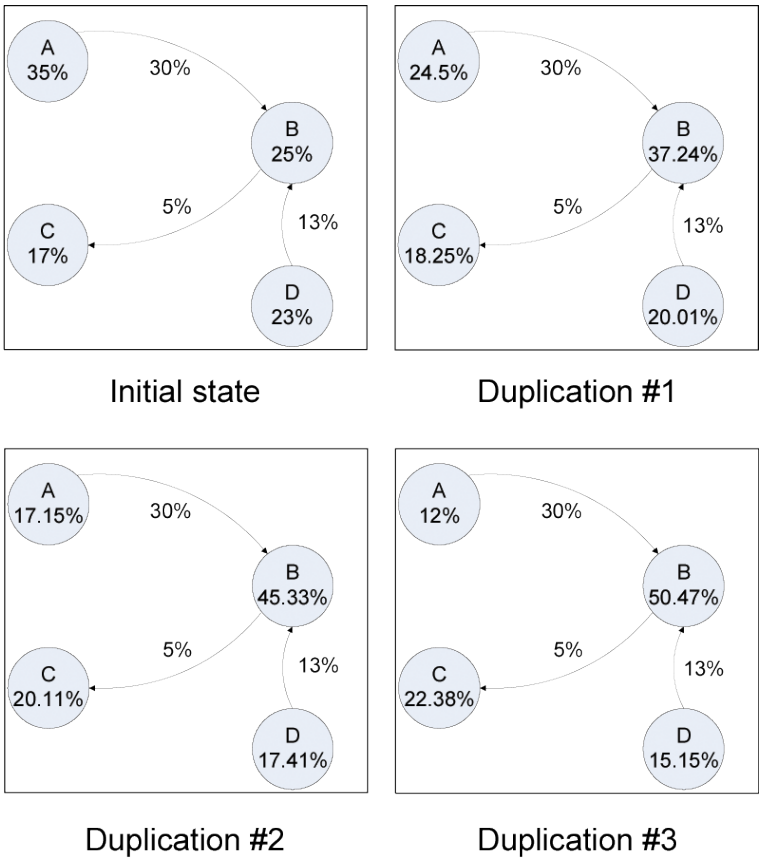

Figure 2.4: Modelization of an example of the Gene Regulatory Network. A, B, C and D are 4 actions with their efficiency coefficient. The transfer coefficients are given by the arrows.

We have implemented this model in Java using a multi-threaded architecture: cells are coded as independent threads. Cells can communicate using the environment and substrate exchanges. We made such a choice because of the development of massive parallel computer architectures such as multi-processor and multi-core machines, increasingly connected in computation grid. This parellelization allows an increase in the number of tasks executed at the same time.

Our model must be able to generate two types of artificial creatures: organs and user defined shapes. The next experiments show that it is possible to accomplish this. The first experiment consists in developing a system able to move susbrates in the enviromnent whereas the second one creates simple shapes like starfish or jellyfish.

To find the creature the most adapted to a specific problem, we use a genetic algorithm. Each creature is coded with a genome composed of three different chromosomes:

- The list of available actions, a subset of the environment possible actions. This list allows the cell to activate or inhibit some actions.
- The action selection system that contains a list of rule to apply actions.
- The gene regulation network that allows cell specification during duplication.

The creature is tested in its environment which returns the score at the end of the simulation. To increase the genetic algorithm power, we use a computational grid parallelized genetic algorithm. This parallelization allows the computation of hundreds of creatures at the same time.

2.3 Experiments

2.3.1 Developing a transfer system

The first experimentation consists of developing a simple organ : a transfer system. In other words, the cell structure must be able to transport substrate from one point to another. To do that, we imagine an environment composed of 2 substrates:

- A red which is the substrate that must be moved by the organism. This substrate has the specificity not to spread in the environment, in order not to impact on the organism work.
- A gray that will be used by the cell as fuel and duplication material.

The cell can perform the following actions:

- duplicate (needs one gray substrate and vital energy),
- absorb or reject substrate (consume vital energy),
- transform one gray substrate in vital energy.

We place 10 red substrate units into a specific cross of the grid (at the top left of the environment) and diffuse gray substrate all over the environment. The creature's score is given by the squared sum of the red substrate distance to the goal point (at the bottom right of the environment). The parameters of the genetic algorithm are:

- selection algorithm: 7 tournament competition with elitism,
- mutation rate: 5%; crossover rate: 65%,

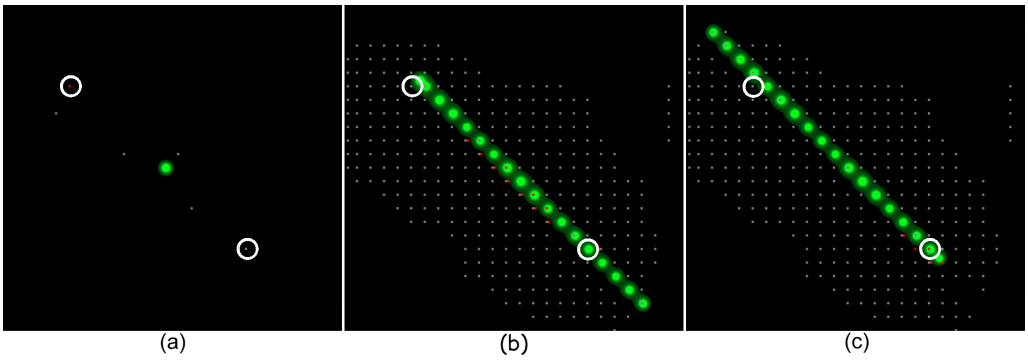

Figure 2.5: Our artificial transfer system. (a) Beginning of the simulation. (b) The creature develops itself to create the structure and begin the substrate transfert. (c) The creature transfers the substrate from the initial state (circle on top left) to the final state (circle on bottom right).

- substitution algorithm: worst individuals,
- population size: 500 individuals,

Figure 2.6 shows the convergence curve of the genetic algorithm. It shows the variation of the minimum, the average and the maximum fitness of the population for each generation. The genetic algorithm's aim is to maximize fitness, which is the creature score. A relevant organism appears quickly. After 3 generations, the organism is able to move the red substrates but not in the right direction. After 10 generations, it is able to move closer to the goal point. The genetic algorithm converges after 22 generations (the average fitness is close to the best).

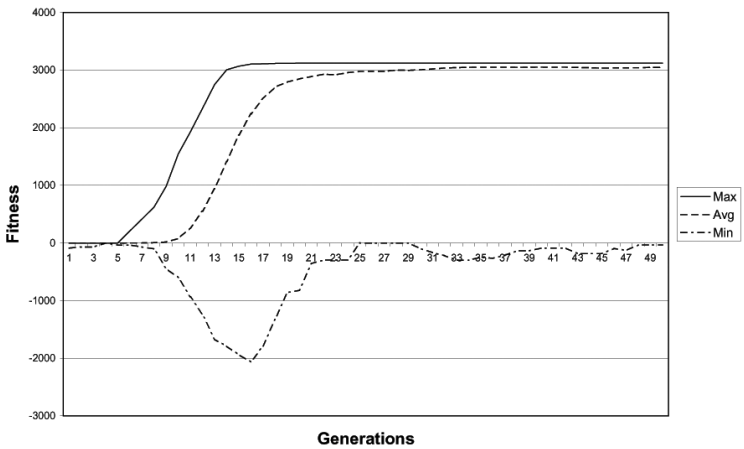

Figure 2.6: Smooth curve of the minimum, average and maximum organism fitness. The genetic algorithm must minimize the sum of the squared distance from the red substrate to the goal point.

Figure 2.5 shows the development of the best organism[1]. We can see that only the cells on the way from the initial point to the end point are created. Moreover, the organism uses absorption and rejection actions to transfer the substrate gradually. Cells that overtake the final point die quickly so as not to interact in the substrate transfer. During the convergence of the genetic algorithm, it is interesting to observe the evolution of the organism strategy towards the best solution. The first step is to learn to survive in the environment, absorbing gray substrate and transforming it in vital energy. The next step is to learn to duplicate in the right direction. Intermediate solution organisms are able to transport the red substrate from the initial point near to the goal. The organism also develops itself throughout the environment, scattering some units of the substrate in the environment. As shown in Figure 2.5, this organism deploys itself only on the best trajectory, decreasing the substrate scattering probability.

2.3.2 Creating simple shapes

In this experiment, we want to generate simple creatures with a user designed morphology. The goal of such an experiments is to simulate the growth of more complex creatures, like those of Sims [Sims(1994)].

To generate these shapes, we needed 5 different substrates :

- *Water* gives energy to cells by transformation ($Water \rightarrow (+30)$). This substrate diffuses in the environment.
- Four different *morphogen* substrates, here named *NW*, *NE*, *SW* and *SE*, show four division directions to cells. These substrates do not diffuse in the environment so as not to interact with the simulation. They are put in place by the designer of the creature.

Associated with these substrates, we have 4 different actions:

- *duplication* consumes energy and one unit of *Water*,
- *water transformation* allows the cell to trigger a transformation of one substrate of *Water* into vital energy,
- *water absorption* allows the cell to pick up water from the environment,
- *apoptosis* allows the cell to autodestruct if it wishes (for example if the cell is not in the desired shape).

[1]Videos of all presented creatures in this paper are available on the website http://www.irit.fr/∼Sylvain.Cussat-Blanc

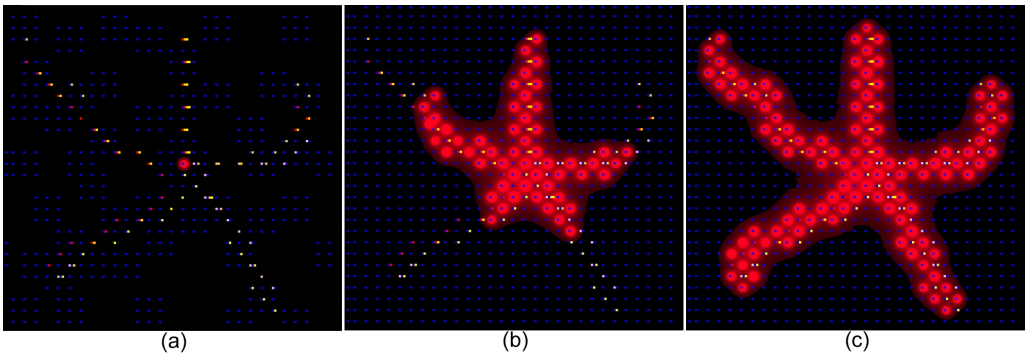

Figure 2.7: The seastar growth. (a) Beginning of the simulation. (b) The starfish develops itself following the morphogens. (c) The starfish stops its growth when the desired shape is obtained.

To obtain the required creature morphology, the genetic algorithm fitness is calucaled after a chosen simulation time and is given by the next simple formula :

- if the cell is inside the desired shape, the fitness value is increased by 2 units,
- if the cell is outside the desired shape, the fitness value is devreased by 1 unit.

The first simple morphology we try to develop using this environment is a starfish[2]. To do that, we place morphogens in the environment to lead the cell divisions. The result of the genetic algorithm is given by Figure 2.7. We can observe that the desired shape is obtained. It is interesting to study the action selection system rules produced by the genetic algorithm:

$$
\begin{aligned}
(SensorNE = 1) &\rightarrow (DuplicateNE)\ (6)\\
(SensorNW = 1) &\rightarrow (DuplicateNW)\ (5)\\
(SensorSE = 1) &\rightarrow (DuplicateSE)\ (4)\\
(SensorSW = 1) &\rightarrow (DuplicateSW)\ (3)\\
&\rightarrow (TransformWater)\ (2)\\
(SensorWater = 1) &\rightarrow (AborbWater)\ (1)\\
&\rightarrow (DoNothing)\ (0)
\end{aligned}
$$

This selection system shows that the genetic algorithm correctly uses the information given by the environment to follow the growth scheme given by the user.

[2]Video : http://www.irit.fr/~Sylvain.Cussat-Blanc

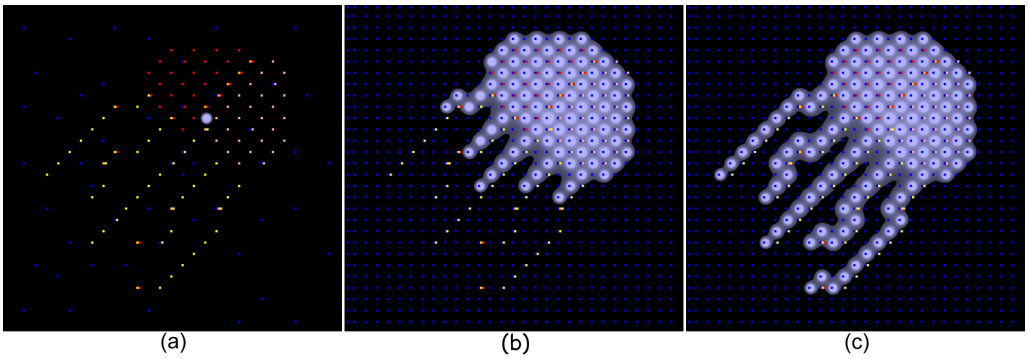

Figure 2.8: The jellyfish growth. (a) Beginning of the simulation. (b) The jellyfish develops itself following the morphogens. (c) The jellyfish stops its growth when the desired shape is obtained.

Moreover, duplications are always prior in relation to other actions to accumulate vital energy without using it. The last remark we can make about these rules is that *Apoptosis* is never used by the organism during growth. The organism assume that morphogens give the correct growth direction.

Observing these rules, we notice that it could be possible to produce all desired creatures with the same genome. Indeed, the rules discovered by the organism allow it to follow any morphogen configuration. To verify the hypothesis, we decided to develop another simple creature: a jellyfish. To do that, we keep exactly the same environment architecture, with the same substrates and the same possible actions, and we only change the morphogen distribution in the environment. Using the starfish genome, we launch the simulation and we obtain the creature[3] shown by Figure 2.8.

[3]Video : http://www.irit.fr/∼Sylvain.Cussat-Blanc

Chapter 3

Genetic Algorithms and Grid Computing

The computation of our creature genotypes take a long time. Because of the time need for each simulation of creature development, the fitness computation of a complete generation composed of thousands genotypes take hours. To reduce the global computation time, we decide to parallelize the genetic algorithm using a computational grid. This chapter is outlined as follows. Section 1 is a state of the art of existing parallel genetic algorithm. It details 3 methods to parallelize genetic algorithm on supercomputer and also show that it is possible to apply it to computational grid. Section 2 presents ProActive[1], a grid programming middleware that provides an infrastructure abstraction of the grid. Section 3 presents different experiments and theirs results of our genetic algorithm parallel version using ProActive and the french computational grid, Grid5000.

3.1 Parallelizing genetic algorithms

Genetic algorithms, by their structure, tend to be easy to parallelize [Cantù-Paz(1997), Herrera et al.(2005), Branke et al.(2004)]. Three main parallelization methods exist but are usually applied with supercomputer. In our work, the supercomputer, generally very expensive, is replaced by a computational grid. In this section, after showing the different parallelization possibilities of a genetic algorithm, we detail already existing parallel genetic algorithm and our way to apply them to artificial life and grid computing.

[1]see http://proactive.inria.fr

3.1.1 Brief review of Genetic Algorithms

This section is a quick review of genetic algorithm. It shows the general functioning of this nature-inspired search method and the different points that can be parallelized. Genetic algorithms follow the next scheme on figure 3.1.

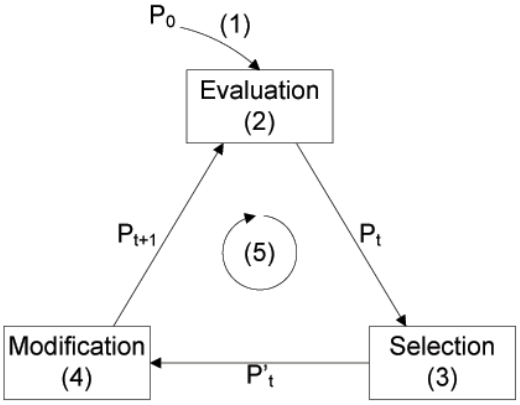

Figure 3.1: Genetic algorithm general functioning scheme.

Initially, a first solution set is chosen, generally randomly, in the problem's search space. Using a fitness function, a score is given to each solution (2). If a sufficient solution is found, the genetic algorithm terminates. A population subset is selected using previous scores (3). Different techniques exist for the selection: bests, tournament, steady-state, etc. To complete the last selected subset, genomes are recombined (4) using two principal operators:

- Crossover. Two genomes are crossed at one or more points to create two new genomes.
- Mutation. One genome is randomly mutated at a random point to create a new genome.

A new population is now available. The algorithm carries on step 2 (5).

Whereas choosing the right parameter values (population size, crossover and mutation rate, fitness quality, etc.) is of key importance to reduce the computation time, a manner to further reduce it consists in parallelizing the genetic algorithm. Because of their total genome independence, 2 points can be parallelized easily: the selection and the genome modifications. Whereas the parallelization of the modification is mostly unprofitable because it does not request heavy computation resources, the parallelization of the selection gives good results. In the next section we present several approaches to reach this parallelization.

3.1.2 Some existing parallel algorithms

Fitness computation parallelization - Master/Worker GA

Using a Master/Worker architecture, the fitness calculation is dispatched on a processor set (figure 3.2). In a classical, mono-processor approach, the algorithm would do the population mixing (using crossovers and mutations) with scores sequentially computed on the same machine. In this approach, the mixing is done on a master computation unit using scores computed by workers deployed on a set of computation units. The general result of the algorithm is therefore exactly the same.

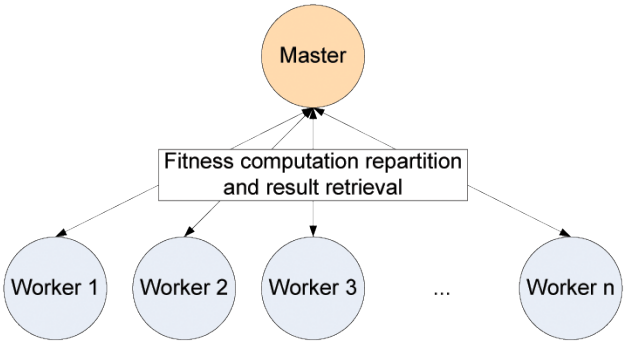

Figure 3.2: A fitness parallelized genetic algorithm allocates fitness computation to different computation units.

Multi-population parallelization - Island GA

The initial population of the genetic algorithm is divided into several sub-populations. Genetic algorithms are applied on each sub-population. The different sub-populations are independent and are computed on different computation units. To help the population mixing between the different sub-populations, migrations of individuals can happen. This avoids the containment onto local optimum of different sub-populations. The individual migration rate is an other genetic algorithm's parameter. Figure 3.3 shows an island genetic algorithm scheme. Different migration architectures are possible such as grid, ring [Sekaj(2004)].

This method heavily decreases the computation unit communication because there are fewer node exchanges. However, the classical genetic algorithm properties are not guaranteed because of the population division that increases the local optimum containment probability. Experiments tend to prove that island GA's results are yet similar to classical GA in quality but need more generation to converge [Bianchini & Brown(1993), Tanese(1989), Neuhaus(1991), Kommu & Pomeranz(1992)].

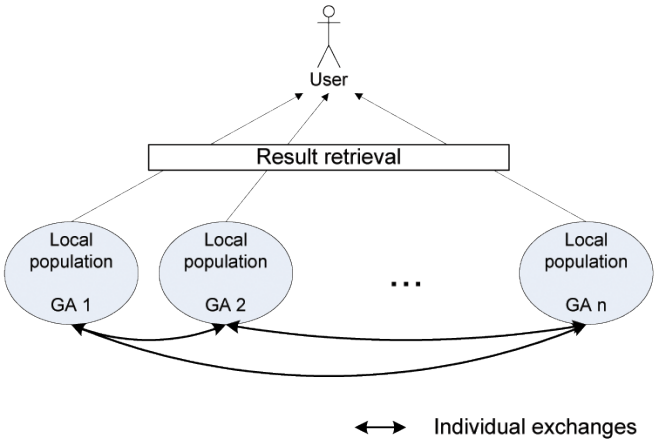

Figure 3.3: An island genetic algorithm creates sub-populations and applies a classical genetic algorithm on each.

Hierarchical genetic algorithm

Combining the two previous methods, one can achieve a further parallelized version of a genetic algorithm. The initial population is divided into sub-populations like in the island GA and each sub-population fitness computation is parallelized [Sekaj(2004), Lim et al.(2007)]. We obtain a hierarchical architecture where each

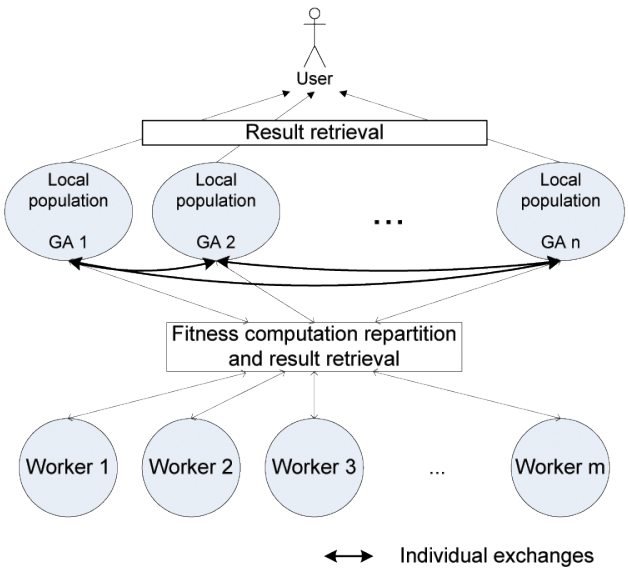

Figure 3.4: A hybrid genetic algorithm create sub-populations and apply a fitness parallelized genetic algorithm on each.

sub-population is a master with different workers that compute fitnesses. Figure 3.4 shows a hybrid genetic algorithm scheme.

Results of such a method are similar to the island GA. Indeed, each sub-algorithm is exactly the same algorithm as the first we present. As we seen before, a genetic algorithm that uses a fitness parallel computation gives exactly the same result as a classical genetic algorithm.

3.1.3 Genetic algorithm on a computational grid

Different implementations of the three last algorithms especially exist on supercomputers. For our researches, we decide to use a computational grid because of their low cost (in comparison to a supercomputer) and the possibility of evolutions (due to the possible heterogeneity). The main difference between grids and supercomputer is that the computation units are separated by a high performance network for grids and by a high performance data bus for supercomputers (Figure 3.5). Because of this network, it is important to consider the data transfer delay.

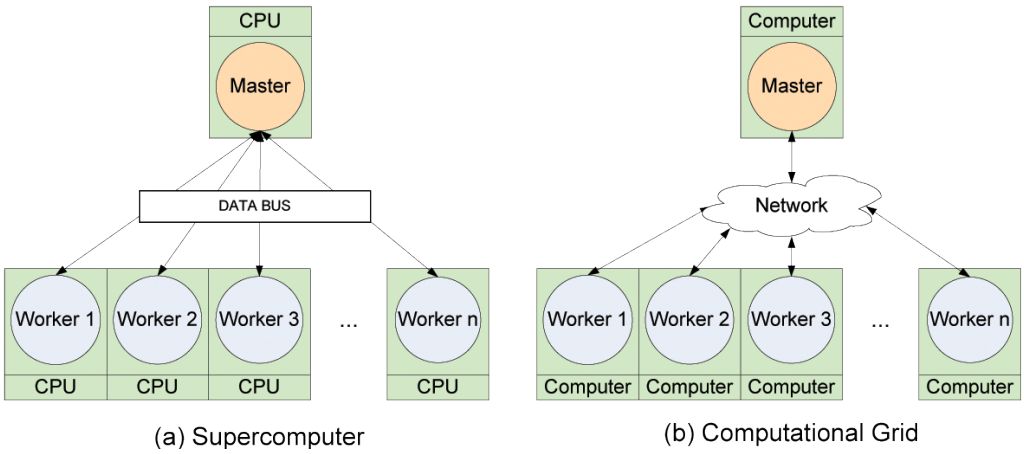

Figure 3.5: In the example of a master/worker GA, a supercomputer (a) communicates using a data bus whereas a computational grid (b) uses a high performance network.

The main goal of this work is to reduce the computation time of our artificial creatures. We decide to apply a Master/Worker algorithm to parallelize our genetic algorithm. This algorithm is well applied to artificial life because creatures genome is small and the fitness computing cost is very important. Because of the small size of the genome, the network restriction forced by a Master/Worker algorithm deployed on a computational grid will not heavily increase the computation time. Moreover, because the properties of a classical genetic algorithm are preserved by the Master/Worker algorithm, the number of generations needed by the algorithm to converge and the final solution quality are exactly the same with or without the parallelization. In this paper, we show that the genome transfer time over the

network is insignificant in relation to the important computation time benefit, in particular in the artificial embryogeny problem.

!!! a replacer plus prcisement !!! Grids gather large amount of heterogeneous resources across geographically distributed sites to a single virtual organization. Resources are usually organized in clusters, which are managed by different administrative domains (labs, universities, etc.). Thanks to the huge number of resources that grids provide, they seem to be well suited for solving very large problems. Nevertheless, grids introduce new challenges such as deployment, heterogeneity, fault-tolerance, communication, and scalability. To not manage all their difficulties, middlewares, such as ProActive, provide a complete abstraction of the grid infrastructure.

In this work, we use a library of Java algorithms called AGMC, developed by our research group, which contains the usual selection and mixing algorithms. In order to implement a parallelized version of the library, we use ProActive grid middleware. The next section describe more in details the ProActive framework, along with its Master/Worker API.

3.2 ProActive and Master/Worker API

ProActive is a grid programming middleware which provides, among others, a Grid infrastructure abstraction using deployment descriptors [Baude et al.(2002)], and an active object model [Caromel(1993)] using transparent futures [Caromel et al.(2006)]. The ProActive framework contains as well a set of toolkits which hide the inner ProActive concepts from the user and provide high-level APIs to well-known class of parallel problems such as :

- Master/Worker
- Branch & Bound
- Skeletons

3.2.1 ProActive's Grid infrastructure abstraction

The ProActive Deployment Framework completely extracts all infrastructure details from the source code [Baude et al.(2002)]. Figure 3.6 shows the general architecture of ProActive.

The first key principle is to *fully* eliminate from the source code the following elements:

- Machine names

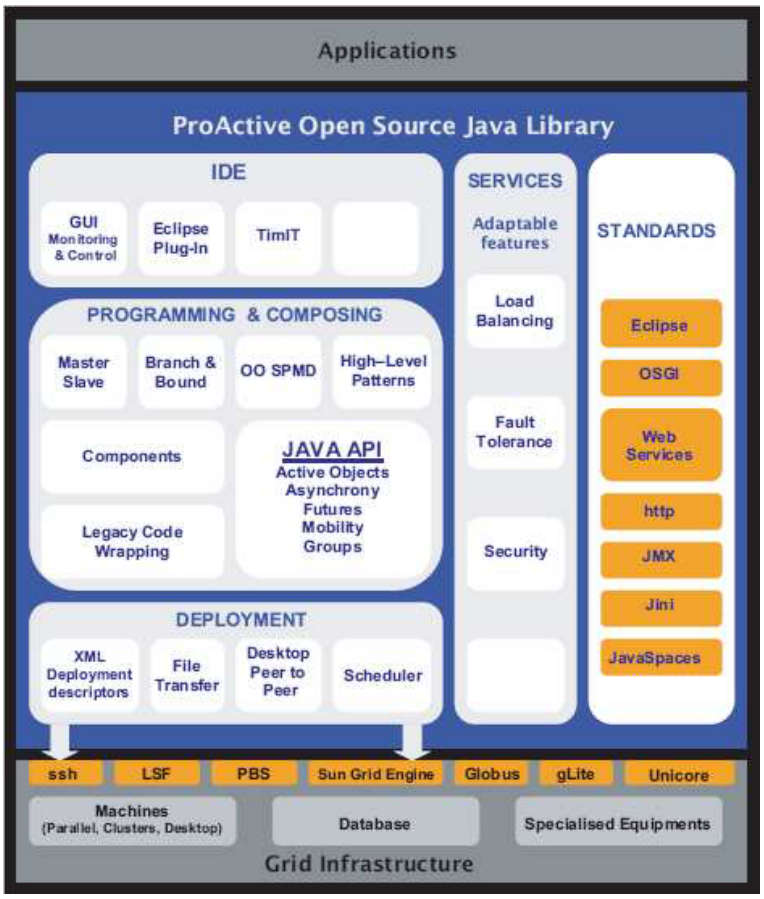

Figure 3.6: ProActive's architecture allows a complete abstraction of the grid infrastructure by providing a set of services.

- Creation protocols
- Registry and lookup protocols
- Communication protocols

The goal of the deployment framework is to deploy any application anywhere without having to modify the source code. The resources acquired through the deployment process are called *nodes*. Nodes are the containers of active objects, and are created by starting the ProActive runtime on the infrastructure resources.

The second key principle is the capability to abstractly describe an application, or part of it, in terms of its conceptual activities.

To summarize, in order to abstract away the underlying execution platform, and to allow a *source-independent deployment* a framework has to provide the following elements:

- An abstract description of the distributed entities of a parallel program or

component.

- An external mapping of those entities to real *machines*, using actual *creation, registry*, and *lookup* protocols.

To answer these principles, the ProActive deployment framework relies on XML deployment *descriptors* to hold the infrastructure configuration. Descriptors introduce the notion of *virtual-node*:

- A virtual-node is identified as a name (a simple string).
- A virtual-node is used in a program source.
- A virtual-node, after deployment, is mapped to one or to a set of *actual ProActive Nodes*, following the mapping defined in an XML descriptor file.

A virtual-node is a concept of a distributed program or component, while a node is a deployment concept that hosts active objects. There is a correspondence between virtual-nodes and nodes which is the relation created in the deployment descriptor: the mapping. This mapping is specified in the deployment descriptor. There is no direct mapping between virtual-nodes and active objects: the active objects are deployed by the application onto nodes related with a virtual-node. By definition, the following operations can be configured in the deployment descriptor:

- The mapping of virtual-nodes to nodes and to Java Virtual Machines.
- The mechanism (protocol) to create or to acquire Java Virtual Machines, such as: local, ssh, rsh, rlogin, lsf, glite, etc.
- The mechanism (protocol) to register or to lookup Java Virtual Machines, such as: RMI, HTTP, RMI-ssh, Ibis, and SOAP.

In the context of the ProActive middleware, nodes designate resources of an infrastructure. They can be created or acquired. The deployment framework is responsible for providing the nodes, mapped to the virtual-nodes, to the application. Nodes may be created using remote connection and creation protocols. Nodes may also be acquired through lookup protocols, which notably enable access to the ProActive Peer-to-Peer infrastructure.

3.2.2 The Master/Worker API

The *master/worker* paradigm is a fundamental and commonly used approach for parallel and distributed applications. In master/worker applications, a single *master*

process controls the distribution of work to a set of identically operating *worker* processes. The master/worker paradigm has been used successfully for a wide class of parallel applications [Pruyne & Livny(1996)] [Everaars & Koren(1997)] [Silva et al.(1999)], and is well suited as a programming model for applications targeted to distributed, heterogeneous "Grid" resources [Berman(1999)].

The ProActive approach to Master/Worker applications is to provide a high-level API which:

- allows the user to simply and freely define tasks that will be executed by the workers.
- internally provides an automatic dispatching of work among the workers.
- provides a simple interface for result gathering
- handles fault-tolerance by redispatching tasks from dead workers.
- is implemented using ProActive and the active-object model.

3.3 Experiments and results

To study the efficiency of the Master/Worker genetic algorithm, we implement a parallel version of our genetic algorithm library using ProActive and specially the Master/Worker API. We deploy the application on the french computational grid, Grid5000. This experimentation's aim is to study the algorithm behavior in different situations:

1. When the computation time is short and the network constraints are important,
2. When the computation time and the network constraints are medium,
3. When the computation time is important and the network constraints are medium.

3.3.1 First Experimentation : OneMax problem

To begin this crop of experiments, we solve the academic OneMax problem. The goal is to create a 30 by 30 binary matrix that contains a maximum of ones. The computation time for such a problem is very short because the fitness function only consists in counting the number of ones in a matrix. At the opposite, the genome's size is important: the genome contains the complete matrix. Moreover, the short

fitness computation time increase the network solicitation. Indeed, the master must continually send new tasks to workers and receive results.

The genetic algorithms parameters for the problem are the following:

- Population size : 100
- Selection method : 7 participants Tournament with elitism
- Crossover rate : 60%
- Mutation rate : 5%

Figure 3.7 shows the results in the computational grid with 1, 4, 8 and 16 CPUs. Each curves' point represents the computation time of one generation.

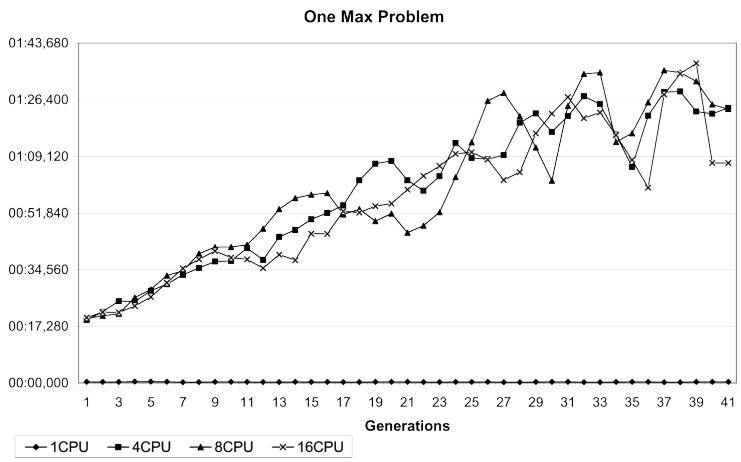

Figure 3.7: Computation time needed to compute 100 generations of the one max problem using 1, 4, 8 or 16 CPU. Because of the low computation needs, the network slows downs the application.

The lower one (with diamonds, near abscissa axis) matches with the mono-processor architecture. The three others represent the parallelized application using 4 CPU (square marks), 8 CPU (triangles) and 16 (crosses). Because of the low computation needs to calculate the fitness value, the grid parallelization of such a problem significantly slows down the application. Curves also show network perturbations. Indeed, the genome sending time over the network is higher than the computation time.

This remark is proved by the equivalence of curves for 4, 8 and 16 CPU. Increasing the number of CPU does not reduce significantly the global computation time.

3.3.2 Second experimentation : Non-invertible matrix

To increase the fitness computation time for each matrix, we try our architecture on a new problem. The aim of this experimentation is to find a non-invertible matrix by computing its determinant. Indeed, to have a non-invertible matrix, its determinant must be equal to zero. The fitness value then corresponds to the distance of the matrix determinant to zero.

Two main methods exist to calculate a determinant:

- Laplace formula: the determinant's computation is equivalent to compute n determinants of a (n-1) by (n-1) matrix (where n is the initial matrix size). The complexity of such a method is $O(n!)$.
- Gauss elimination: the aim is to combine lines or rows to create the maximum of zeros in the matrix. The final computation is done with Laplace formula. Because of the number of zero in the matrix, the computation is accelerated. The complexity of Gauss elimination is $O(n^4)$.

To increase the computation time "artificially", we decide to implement Laplace's method with an 11x11 matrix. The mean computation time for a determinant of such a matrix is about 18 seconds[2].

To find our non-invertible matrix with a genetic algorithm, genomes are simply composed by a crop of integers belonging to the interval $[-10, 10]$. The genetic algorithm's parameter values were obtained by trying different values with a classical genetic algorithm. They correspond to the fewer generation number until the convergence. Theirs parameters are the following:

- Population size : 2000
- Selection method : 7 participants tournament with elitism
- Crossover rate : 55%
- Mutation rate : 7%

Figure 3.8 represents smoothed curves for this experimentation. You can note that the ordinate axis is graduate using a logarithmic scale for the time. For this experimentation, we use 1, 50 and 100 processors in the grid. In the three cases, the computation time for each generation is nearly constant generation after generation. The computation time is sufficient to compensate the network exchange duration. Moreover, the global computation time is proportional to the number of CPU.

[2]Experiment performed using a 2.33GHz processor with 4GB of memory.

Indeed, with 1 CPU, a generation takes on the average 243 minutes to be calculated, with 50 CPU, it takes 4.5 minutes (54 times less than 1 CPU) and with 100 CPU, it takes 3 minutes (that is to say 81 times less than with 1 CPU). The time taken by the network exchanges for this experimentation is not very important in comparison to the time won.

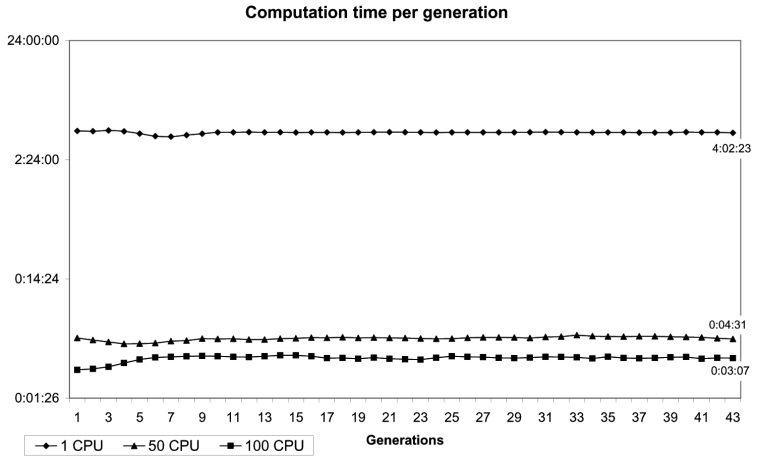

Figure 3.8: Computation time needed for each generation (smoothed curve). The grid parallelization proportionately reduces the generation computation time.

3.3.3 Third experimentation : Artificial embryogeny model

We are going to apply our parallelization method on the most interesting problem for us: the generation of artificial creatures able to grow in a virtual environment.

In this experimentation, we use the substrate transfer system presented in chapter 2 All cells' specifications are coded in its genome. The genome size is about 19 kbytes and the simulation duration vary from a couple of seconds (when the organism is unable to do anything) to 120 seconds (when the organism develops itself in the environment and performs the asked action). Due to this important variation, the load balancing given by ProActive will optimize the task repartition amongst the workers. The genetic algorithm parameters for this problem are:

- Population size : 750
- Selection method : 7 participants tournament with elitism
- Crossover rate : 65%
- Mutation rate : 5%

Curves in figure 3.9 represent the computation time for each generation for one and 50 CPU. Note that the ordinate axis is graduated using a logarithmic scale for the time.

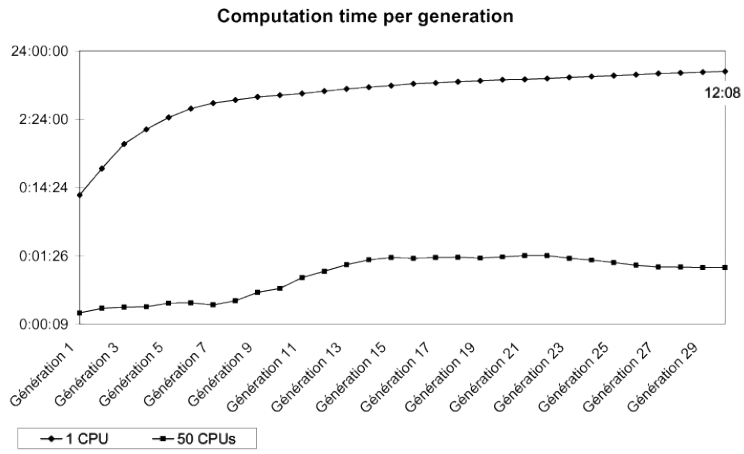

Figure 3.9: Computation time needed for each simulation generation (smoothed curve). The grid parallelization reduces the calculation explosion due to the creature evolution.

First of all, the two curves increase generation after generation because the computation time to calculate the creature fitness globally increases. Indeed, in the first generations, a lot of creatures are unable to survive. Then, they die immediately, the simulations stop and the calculation time for such a simulation is very short (about one second). But, generation after generation, creatures evolve to use the environment's resources and to develop themselves. Then, the computation time to evaluate each creature grows up to 120 seconds.

The second interesting thing is that the global computation time is highly decreased. The network solicitation is lower than in the one max problem and the computation time for each creature is sufficient to obtain good results even at the first generations. The difference between the two curves increases generations after generations. This computation time reduction is due to the load-balancing provided by ProActive. Workers always have a fitness to compute even if the last one was very short to compute (because the creature was unable to survive in its environment for example).

3.4 Discussion

In this chapter, we present a parallel version of genetic algorithms based on grid computing. This version is especially interesting for our specific artificial embryo-

geny problem. Indeed, in such a problem, genomes have relatively small sizes and important computation needs. To parallelize this problem, we use a Master/Worker genetic algorithm. Results given by such an algorithm are very promising. The most interesting result is that the generation's computation time explosion due to creature evolution is reduced thanks to load-balancing. It significantly reduces the global computation time. Yet, the use of a computational grid implies some restrictions: it is almost impossible to repeat the same experimentation many times. Indeed, in a grid like Grid5000, a reservation system has been introduced to have a set of computers. It is hard to have exactly the same set two times on the bounce. Moreover, the network behaviour between two experiments can be very different and can impact the results.

Another problem is the master memory explosion observed with a very large population. Whereas a genetic algorithm uses an important quantity of memory, the use of ProActive middleware does not improve the problem. Indeed, each genome is encapsulated in a task to be sent to worker. This memory explosion can be reduced by an island genetic algorithm but the convergence time can increase [Bianchini & Brown(1993)]. It could be interesting to implement an island algorithm using ProActive and to deploy it on Grid5000 to compare it with our Master/Worker genetic algorithm. We think that the gain provided by the network solicitation reduction of such an algorithm will not reduce the computation time. Another advantage of this algorithm is the memory repartition due to the division into sub-populations. Because the population is distributed amongst different computers, the memory problem would then be reduced.

Chapter 4

Conclusion and future works

We propose a model of cellular development. This model is based on a marked simplification of natural development. We ignore the physics rules and the atomic and molecular interactions to focus on the cell abilities. Using a genetic algorithm and specific environment, we create a organism able to develop different organs with different functions. As we have shown during experiments, this model can produce various creatures with very different morphology or different functions.

We also present a parallel version of our genetic algorithm library. It is particularly well adapted for artificial embryogeny where the fitness computation time is mostly long. In the artificial embryogeny model we present, a parallelization level already exists. Because of the massive cell management of such a model, cells are parallelized using a multi-threaded architecture. After some experiments, we show that a fitness parallelization gives good results thanks to a computation grid. Because each grid node has a multi-processor or multi-core architecture, it allows the simulation to be optimized.

The continuation of this work presents a wide field of development. Developing new organs can be interested. For example, the next one could be an organ able to harvest different substrates and transform them into vital energy and dispose wastes at a specific position. Using different types of such an organ, the wastes of one used as energetic substrate by another, we will produce a complete creature composed of different organs. The different organs will be connected using the presented transfer system.

Another improvement may concern shape generation. For the moment, we use four different morphogens to obtain the creature morphology. We think that with only one morphogen and only giving the development main line, we could obtain the same creature and have an organ that develops itself corectly to produce this morphogenetic substrate. For example, in the case of the starfish, we could have

a transfer system that moves the morphogenetic substrate from the center of the environment to the five branches of the starfish. In a second stage, the starfish will grow using the morphogen distribution.

A remark we can make when we watch the starfish growth is that all the branches do not grow at the same speed. The same fact can be noticed in jellyfish growth, where the bell-shape grows too fast in comparison with tentacle development. An idea to control shape development is to calculate fitness at different moments of the simulation. The best creature will then be the one that produces the best shape at each checkpoint.

A final development path is the abstraction of this model. Starting from a unique cell, we grow shapes like the starfish or the jellyfish presented in the paper and, after a cell regroupement to different limbs, we put the creature in a physical simulator to make it move. The creature movements could be generated, for example, by a neural network, just like in Sims' works [Sims(1994)]. This abstraction will allow us to have a complete creature development, from single cell to a creature able to move in its environment.

Acknowledgments

ProActive is a middleware (part of the ObjectWeb consortium, with Open Source code) for parallel, distributed and multi-threaded computing. It is provided by OASIS team in INRIA Sophia Antipolis (see http://proactive.inria.fr)

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https:// www.grid5000.fr)

List of Figures

Bibliography

[Banzhaf(2003)] W. Banzhaf. Artificial regulatory networks and genetic programming. *Genetic Programming Theory and Practice* pp. 43–62 (2003).

[Baude et al.(2002)] F Baude, D Caromel, L Mestre, F Huet, & J Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pp. 93–102, Edinburgh, Scotland. IEEE Computer Society (2002).

[Berman(1999)] Francine Berman. High-performance schedulers. In *The grid: blueprint for a new computing infrastructure*, pp. 279–309. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999).

[Bianchini & Brown(1993)] R Bianchini & C Brown. Parallel genetic algorithms on distributed-memory architectures. *Technical Report (revised version), University of Rochester* (May 1993).

[Branke et al.(2004)] J Branke, A Kamper, & H Schmeck. Distribution of evolutionary algorithms in heterogeneous networks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 923–934 (2004).

[Cantù-Paz(1997)] E Cantù-Paz. A survey of parallel genetic algorithms. *Technical report 95004, Illinois Genetic Algorithms Laboratory, Urbana, IL* (1997).

[Caromel(1993)] D Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM* 36(9):90–102 (1993).

[Caromel et al.(2006)] D Caromel, C Delbe, A di Costanzo, & M Leyton. Proactive: an integrated platform for programming and running applications on grids and p2p systems. *Computational Methods in Science and Technology* 12 (2006).

[Chavoya & Duthen(2007)] A. Chavoya & Y. Duthen. Evolving an artificial regulatory network for 2d cell patterning. *Proceedings of the 2007 IEEE Symposium on Artificial Life* pp. 47–53 (2007).

[Cussat-Blanc et al.(2007a)] S. Cussat-Blanc, H. Luga, & Y. Duthen. A developmental model to simulate natural evolution. *International Conference on Computer Graphics and Artificial Intelligence 3IA* (2007a).

[Cussat-Blanc et al.(2007b)] S Cussat-Blanc, H Luga, & Y Duthen. Using a single cell to create an entire organ. *International Conference on Artificial reality and Telexistance (ICAT)* (2007b).

[Davidson(2006)] E. H. Davidson. The regulatory genome: gene regulatory networks in development and evolution. *Academic Press* (2006).

[Dawkins(1986)] R. Dawkins. The blind watchmaker. *Longman Scientific & Technical* (1986).

[de Garis(1999)] Hugo de Garis. Artificial embryology and cellular differentiation. In editor Peter J. Bentley (ed.), *Evolutionary Design by Computers*, pp. 281–295 (1999).

[Dellaert & Beer(1994)] Frank Dellaert & Randall Beer. Toward an evolvable model of development for autonomous agent synthesis. In *Artificial Life IV, Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, Cambridge, MA. MIT press (1994).

[Dittrich et al.(2001)] Peter Dittrich, Jens Ziegler, & Wolfgang Banzhaf. Artificial chemistries - a review. *Artificial Life* 7(3):225–275 (2001).

[Eggenberger Hotz(2004)] Peter Eggenberger Hotz. Asymmetric cell division and its integration with other developmental processes for artificial evolutionary systems. In *Proceedings of the 9th International Conference on Artificial Life IX*, pp. 387–392 (2004).

[Everaars & Koren(1997)] C T H Everaars & B Koren. Using coordination to parallelize sparse-grid methods for 3D CFD problems. In *219*, p. 23. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X (1997).

[Fukunaga et al.(1994)] A. Fukunaga, J. Marks, & J. T. Ngo. Automatic control of physically realistic animated figures using evolutionary programming. *Proc. of Third Annual Conference on Evolutionary Programming* pp. 76–83 (1994).

[Garcia Carbajal et al.(2004)] S. Garcia Carbajal, M. B. Moran, & F. G. Martinez. Evolgl: Life in a pond. *Artificial Life XI* pp. 75–80 (2004).

[Gardner(1970)] M. Gardner. The fantastic combinations of John Conway's new solitaire game life. *Scientific American* 223:120–123 (1970).

[Herrera et al.(2005)] J Herrera, E Huedo, R S Montero, & I M Llorente. A grid-oriented genetic algorithm. *in Advances in Grid Computing - EGC 2005* pp. 315–322 (2005).

[Holland & Reitman(1978)] J. H. Holland & J. S. Reitman. Cognitive systems based on adaptive algorithms. *Pattern-Directed Inference Systems* (1978).

[Hutton(2007)] Tim J. Hutton. Evolvable self-reproducing cells in a two-dimensional artificial chemistry. *Artificial Life* 13(1):11–30 (2007).

[Kauffman(1969)] S. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theorical Biology* 22:437–467 (1969).

[Kommu & Pomeranz(1992)] V Kommu & I Pomeranz. Effect of communication in a parallel genetic algorithm. In *ICPP (3)*, pp. 310–317 (1992).

[Komosinski & Ulatowski(1999)] M. Komosinski & S. Ulatowski. Towards a simulation of a nature-like world creatures and evolution. In *ECAL '99: Proceedings of the 5th European Conference on Advances in Artificial Life*, pp. 261–265, London, UK. Springer-Verlag (1999).

[Lassabe et al.(2007)] Nicolas Lassabe, Herv Luga, & Yves Duthen. A New Step for Evolving Creatures. In *IEEE-ALife'07*, pp. 243–251. IEEE (2007).

[Lim et al.(2007)] Dudy Lim, Yew-Soon Ong, Yaochu Jin, Bernhard Sendhoff, & Bu-Sung Lee. Efficient hierarchical parallel genetic algorithms using grid computing. *Future Gener. Comput. Syst.* 23(4):658–670 (2007).

[Neuhaus(1991)] P Neuhaus. Solving the mapping problem - experiences with a genetic algorithm. In *PPSN I: Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, pp. 170–175, London, UK. Springer-Verlag (1991).

[Ono & Ikegami(1999)] Naoaki Ono & Takashi Ikegami. Model of self-replicating cell capable of self-maintenance. In *ECAL '99: Proceedings of the 5th European Conference on Advances in Artificial Life*, pp. 399–406, London, UK. Springer-Verlag (1999).

[Pruyne & Livny(1996)] J Pruyne & M Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Future Generation Computer Systems* 12(1):67–85 (1996).

[Rasmussen et al.(2003)] Steen Rasmussen, Liaohai Chen, Martin Nilsson, & Shigeaki Abe. Bridging nonliving and living matter. *Artificial Life* 9(3):269–316 (2003).

[Sekaj(2004)] Ivan Sekaj. Robust parallel genetic algorithms with re-initialisation. In Xin Yao, Edmund Burke, Jose A. Lozano, Jim Smith, Juan J. Merelo-Guervós, John A. Bullinaria, Jonathan Rowe, Peter Tiňo Ata Kabán, & Hans-Paul Schwefel (eds.), *Parallel Problem Solving from Nature - PPSN VIII*, vol. 3242 of *LNCS*, pp. 410–419, Birmingham, UK. Springer-Verlag (2004).

[Silva et al.(1999)] Luís Moura Silva, Victor Batista, Paulo Martins, & Guilherme Soares. Using mobile agents for parallel processing. In *DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications*, p. 34, Washington, DC, USA. IEEE Computer Society (1999).

[Sims(1994)] K. Sims. Evolving 3d morphology and behavior by competition. *Artificial Life IV* pp. 28–39 (1994).

[Stewart et al.(2005)] Finlay Stewart, Tim Taylor, & George Konidaris. Metamorph: Experimenting with genetic regulatory networks for artificial development. In *ECAL '05: Proceedings of the 8th European Conference on Advances in Artificial Life*, pp. 108–117 (2005).

[Tanese(1989)] R Tanese. Distributed genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pp. 434–439, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. (1989).

[Ventrella(1998a)] J. Ventrella. Attractiveness vs efficiency (how mate preference affects location in the evolution of artificial swimming organisms). *Artificial Life VI* pp. 178–186 (1998a).

[Ventrella(1998b)] J. Ventrella. Designing emergence in animated artificial life worlds. *Internationnal Conference on Virtual Worlds* pp. 143–155 (1998b).