Bio Molecular Engine: A bio-inspired environment for models of growing and evolvable computation

A. Gallini
Dept. of Computer Science, Systems and Communication
University of Milan-Bicocca,
Milan, Italy
PHONE: +39-02-64487843
alberto.gallini@disco.unimib.it

C. Ferretti
Dept. of Computer Science, Systems and Communication
University of Milan-Bicocca,
Milan, Italy
PHONE: +39-02-64487819
ferretti@disco.unimib.it

G. Mauri
Dept. of Computer Science, Systems and Communication
University of Milan-Bicocca,
Milan, Italy
PHONE: +39-02-64487828
mauri@disco.unimib.it

ABSTRACT

Evolutionary computation has been often used by computer scientists to evolve the morphologies and control systems of artificial life. Artificial 'brains', behaviour strategies, methods of communication, distributed problem solving and many other topics are commonly explored by using genetic algorithms and other evolutionary search techniques. We think that this approach may provide the general guidelines to efficiently manage and "design" computation on large and homogeneous lattices of simple, asynchronously interacting processing elements. Because of their structural simplicity, this kind of substrates will be suitable architectural models for computational machines based on molecular scale devices. In this paper we present an environment named Bio-molecular Engine (BME), in which different substrates can be simulated and used as "artificial worlds" where computational entities can rise, grow and evolve. In particular we discuss how to use a grid to evolutionary find a good solution to a well defined design issue: how much parallelism is good for a given problem computed in our environment.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation – *unbounded action device, computability theory.*

General Terms

Algorithms, measurement, performance, design, experimentation.

Keywords

Simulation software, evolutionary algorithms, scalability, computer architectures, cellular arrays.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Genetic and Evolutionary Computation Conference (GECCO)'05, June 25-29, 2005, Washington, DC, USA.

Copyright 2005 ACM 1-59593-097-3/05/0006…$5.00.

1. INTRODUCTION

The evolutionary approach draws inspiration from biology. Another meeting point between computation and biology is "parallelism", since new computing devices could be built where many simple processing elements work in parallel, like cells in organisms.

Motivation for this modelling choice is also given by new technological challenges arising from nanotechnology: it will mostly allow very small but simple processing elements, thus requiring a high level of parallelism to obtain more computational power (see [1]-[4]). We present here a biologically inspired parallel computing architecture, and a simulator for it. The simulator features a high level of flexibility in the definition of the simulated architecture, thus allowing to experiment with different design choices. The basic assumptions being to have many small connected processors, working in parallel. Anyway a high level of parallelism is not always exploitable, depending on the algorithmic properties of the computed problem. Moreover, it could also happen that a single parallelizable problem, at different levels of parallelism, provides different performances on a given computational architecture. Thus, a design problem arises: we deal with a trade-off between assigning long computations to few processors, and assigning simple computations to many communicating units. We need to map a computation onto parallel devices in such a way that we obtain the best performance under that trade-off. We tried to use evolutionary approach to this problem, and the resulting optimizing algorithm was computed in our simulated model.

Section 2 describes the basic features of the simulator; Section 3 introduces the biological reference concepts for this architecture, while Section 4 gives more details on the resulting implementation; Sections 5 and 6 deal with the design problem we considered: the first one gives the analysis, and the second one presents an evolutionary approach to it; finally, Section 7 draws some conclusions and discusses perspectives.

2. THE ENVIRONMENT

The BME simulator provides a biologically inspired environment where *execution entities* rise and grow on a grid of homogenous virtual processing elements in order to perform computations. These entities are completely *autonomous*: during their life they autonomously search and exploit resources to achieve their tasks, there is no hierarchically superior unit to control them or manage system resources.

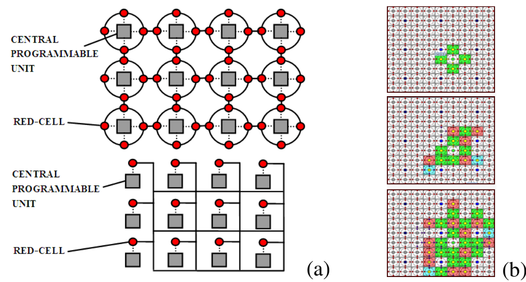


Figure 1. (a) Virtual hardware lattices. (b) A growing computation.

The execution entities define regions on a space-extended virtual hardware whose features can be defined by user. The virtual hardware is graphically represented as a lattice (see Figure 1) with a regular 2D structure of processing elements or *cells*; this choice was only to make representation simple, but there's no limit to the interconnection degree and there are no constraints to the network topology and to the number of elements, except for those imposed by the amount of resources. The users can specify a lattice with specific features (e.g. topology described in [16]) simply extending general (Java) classes of the simulator.

The execution entities are composed by cells and can be considered as artificial organisms with specific functions. Their behavior is determined by a program whose code (the organism's genome) is enclosed in every cell of the organism (see section 3).

An artificial organism rises from a single cell and expands itself specializing neighbor cells on the lattice: this growth is a sort of self-organizing process in which every cell finds its right place depending on genome instructions and environmental conditions. Cell *specialization* consists in copying the genome from a cell of the organism to a non specialized cell and marking a code subsection as an *active gene* that determines the services the new cell will provide the environment with.

Every cell is composed by a central programmable unit for manipulation of small amount of information and by a set of one or more red-cells. A red-cell has to control the interactions with the external environment[1]. A red-cell receives a set of packets and discriminates (in function of the internal state of the central unit) which message has to be absorbed and what has to be routed for different destinations.

A cell can detect an internal fault[2] and excludes itself after sending a recovery signal; another (back-up) cell will provide services of damaged cell. Therefore, a region of lattice has a certain degree of fault-tolerance, provided by a set of regularly distributed spare cells.

Lattice components (central units and red-cells) are implemented with a thread so that every high-level hardware unit is represented by a different execution flow: cell-threads perform computation and red-cell-threads perform communication by means of messages exchange. Every cell encloses an instance of a user definable interpreter which specifies cellular operations and directly executes the code; each red-cell manages a set of queues for local packets storage and together with other red-cells composes a net in which message passing is performed using the producer-consumer paradigm. User-definable routing rules are associated to red-cells in order to define their behaviour. Every packet is marked with a precise destination signature, which is used by red-cells to perform local choices; they take routing decisions, after reading the destination mark, message routing header and values contained inside available user definable routing structures (i.e. local look-up tables).

BME is a highly flexible instrument and provides a suitable environment where the user, after specifying virtual hardware features (i.e. connectivity and routing rules) and the application to be executed, can test different solutions for *spatial distribution of computation*.

3. BIO-INSPIRED COMPUTATION

3.1 Computation hierarchy

The lattices provide a static substrate on which it's possible to create artificial organism with specific functions. An artificial BME organism has a hierarchic structure that recalls organization of biological organism (like in [13]-[15]). The basic element is the cell with its own specialization: the pair <*cell*, *specialization*> is called *BME-thread* or simply *thread*. A thread in BME is, at the same time, a logical entity and a physical entity, because it identifies a cell with a precise configuration. A set of cells with the same specialization constitutes a *tissue*. Two or more different tissues grouped together represent an artificial *organ* or *process* that is a structure providing a larger number of services. An *organism* or *program* is given by grouping different organs and performs a task that provides results for a certain computation .

In nature every cell of a multi-cellular organism contains the entire genetic code that completely describes the structure of the organism. In BME this "universality" of the cells is considered a useful redundancy (because of its own positive repercussion on fault-tolerance aspects), but it's maintained at the *process* level, since the programming-genetic-code for an entire system could be too large. In spite of its process-universality, the cell only uses the *module* that defines its *activity*. This module is identified with the expression of *method* or *active gene* and determines the set of operations a cell provides the environment with.

A BME-thread is the analogue of a biological specialized cell: during its life, it elaborates a set of data in function of the code that specifies its behaviours. The result of this elaboration is the analogue of proteins. Like a cell with a given specialization produces a well defined class of proteins, so a thread elaborates its input in a precise way dictated by its code. The structural specifications of a cell and, as a consequence, programming paradigms exploited to define cell code, determine the complexity of services provided by a single cell.

The biologically inspired hierarchy provides general guidelines about organization of computation in BME environment, but nothing is said about functional distribution of computation: How, can a problem be decomposed into simpler tasks? And, what is the most suitable nature of cellular level operations for it?

BME doesn't give any direct answer to these questions, but it has been created to find them through experimentation.

When users want to create organisms on the BME lattice, they have to start from a data-flow graph representing the desired

[1] "External environment" is an expression referred to a cell and means "everything outside that cell".

[2] Cellular Damages are inducted by user with a mouse click on the desired cell picture in the lattice graphical representation window.

application (even obtained by means of parallelizing compiler like in [5],[6],[22] or in [7],[8]): first a precise specification of cellular operations set is needed and then it is possible to build a data-flow graph. Afterwards, the user owns all the instruments to write code and give it in input to the simulator.

Currently the user can test different distribution degree for a given application by modifying code, but BME can be exploited to develop evolutionary approaches for a *self-organizing computation* (as we will show in section 6) creating populations of artificial organisms able to autonomously evolve towards better spatial resources usage: "evolution" applied to computation will have a positive impact on *scalability*, as spatial allocation policies dynamically change in function of environment properties, first of all dimensions and topological constrains.

3.2 Cellular Life-cycle

When there's no loaded process, lattice appears like a field of *not existent cell* with a uniform distribution of *not existent staminal cell*; both these kinds of cells don't have a specialization so they are potential resources. A cell becomes "existent" whether it acquires a specialization (*eukaryote cell*) – so a thread rises - or becomes an *active staminal* (a back-up cell ready to assume a specialization of damaged cell)[3].

The life of a process starts from a single cell with a particular

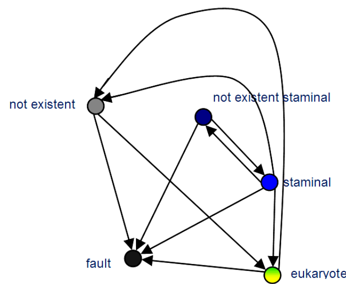


Figure 2. Cell life-cycle.

specialization given by a module called "Main". From this BME-thread, the artificial organism begins to grow by "cellular division", that is performed by two particular kinds of *specialization packets* emitted by cells:

- *run* packets
- *call* packets

The thread generated by a "run" packet is concurrent to the thread which has emitted the specialization packet, while a thread which has emitted a "call" packet, waits for a return value coming from the new "called" thread (i.e. caller and callee interacts by extended rendezvous [21]). A computation acquires spatial resources by means of specialization packets: they specialize the first not existent cell found on their path, as determined by routing rules and conditions of the environment (i.e. previously specialized cells and their position).

Specialization packets are emitted by a cell when it executes special "run" and "call" primitives (see section 4) contained in its genome. These primitives require the name of a module as input in order to generate a cell which will execute it, e.g.:

```
run moduleForSum  or  call moduleForSelection
```

[3] Staminal Cells are currently not used, because fault-tolerance policies are currently work in progress.

So "run" and "call" primitives trace relationships between two modules of the code ("caller" and "callee"). It is possible to draw a graph that depicts the logical structure of a process by representing modules as nodes and relationships as edges; figure 4-b shows an example of a process with a logical tree structure.

Threads interact by means of *data packets*; differently from specialization packets, their routing is intended as a searching activity based on local information about disposition of resources (no cell completely knows the process morphology). When a message arrives next to the destination cell it is absorbed like biological cells absorb external resources through their membrane. Data are transformed by cells on the base of cellular specialization code, i.e. its active gene. In general, threads interactions are not constrained by modules relationships.

A BME-thread, in the course of its life, can assume different internal states. It can apply operations on data (*run* state) or invoke other methods to request other services. A thread could *wait* data from other threads or could be in a *pending* state, waiting for a return value of a thread it has invoked via a "call" command.

If a thread terminates correctly then it assumes *end* state otherwise, if it has detected an internal damage[4] it assumes *error* state.

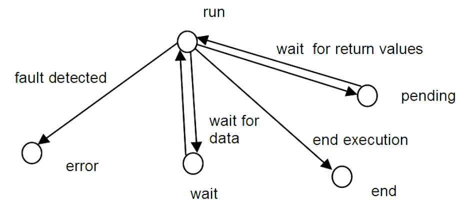


Figure 3. Cell internal state.

4. IMPLEMENTATION

4.1 Virtual devices

Java threads are used to implement the high level hardware units of lattices: cells and red-cells. Red-cells enclose a certain (user definable) number of queues controlled by a *queues manager* that inserts incoming packets in the queues and extracts from them outgoing packets in FIFO mode. When a packet is selected by queues manager, its destination (one of neighbour red-cells or the current cell) is chosen by applying a user-selected routing rule.

Cell-threads implement a very simple machine capable of few actions or *cellular primitives*. These primitives are represented by some methods of class "Cell". Associated to the cell there is an interpreter employed to execute operations on data and, since it is user-definable, a cell virtually supports any kind of programming language. Yet, every interpreter has to internally define the cellular primitives in order to provide the programmer with a usable cell interface and, as a consequence, with BME environment.

4.2 Cellular functional code

An interpreter for the cells has to accept a language that includes base primitives and defines a set of operations on data. Base primitives are concerning in *growth*, *data*, *specialization* and

[4] Internal damages are inducted by user with a mouse click on a cell.

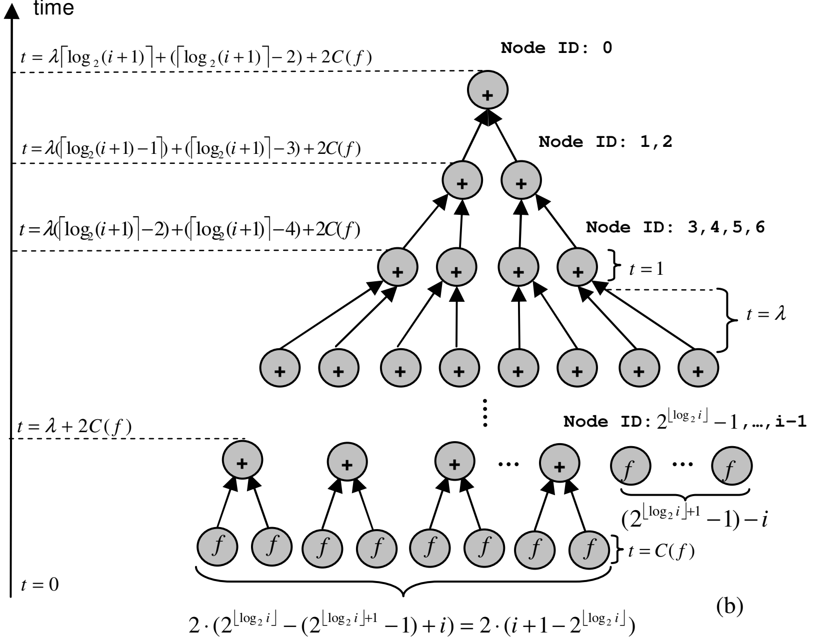
```
Main
[null]
[null]
BeginMethod
$nodeID = 1
run FX $nodeID
$nodeID = 2
run FX $nodeID
wait $result_left_subtree
wait $result_right_subtree
$result_left_subtree + $result_right_subtree
out $result_left_subtree
EndMethod

FX
[IN]
[null]
BeginMethod
Every FX cell acquires  node ID used to discriminate internal nodes
from leaves:
get $nodeID
Every FX cell knows  tree features:
$numInteranlNodes  = 100
$maxInNodeID = $numInNodes
$maxInNodeID - 1

if (actual node is internal)
begin
   current  node generates offspring,
   $nodeID_left = $nodeID
   $nodeID_left * 2
   $nodeID_left + 1
   $nodeID_right = $nodeID_left
   $nodeID_right + 1
   run FX $nodeID_left
   run FX $nodeID_right
   waits results,
   wait $Xi
   wait $Xj
   performs  SUM of offspring results.
   $Xi + $Xj
end

if (actual node is a leaf)
begin
    current  node performs f(x).
    $Xi = f(x)
end

if ($nodeID < 3)
begin
    if current  node is a child of the root  then sends result to the Main,
    send Main null $Xi
end

if ($nodeID > 2)
begin
     else sends result to its parent.
    send FX $nodeID_parent IN $Xi
end
EndMethod
```

(a)

A NODE GENERATES TWO OFFSPRING

COMMENTS

PSEUDO-CODE



MAIN (ROOT)

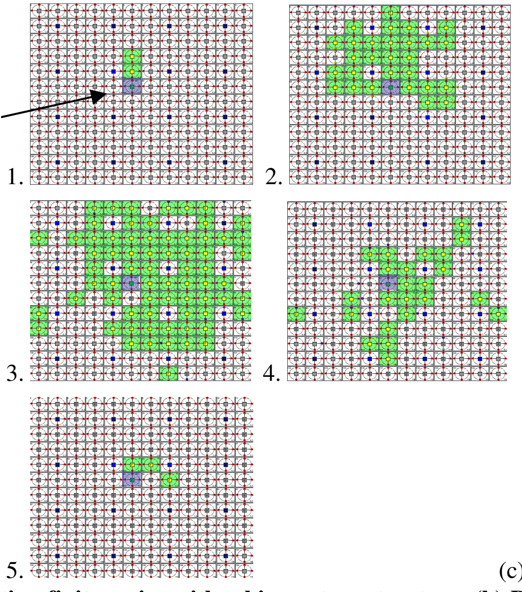


Figure 4. (a) BWL program that define the cellular genome for calculating finite series with a binary tree structure. (b) Binary tree structure of the finite series and complexity analysis. Arrows on the edges show data flow through the nodes. (c) Execution of the finite series program with a binary tree structure on BME using a non-deterministic routing rule. The tree grows on the lattice, starting from the root; every node, after performing local computation, returns its result to the environment and expires.

local memory access. Table 2 briefly describes the meaning of some base primitives.

Table 2. Cellular base primitives

primitive	area	Action
CALL	*growth*	“call” packet emission
RUN	*growth*	“run” packet emission
SEND	*data comm.*	“data” packet emission
WAIT	*data comm.*	cell waits for incoming data
GET	*specialization*	A cell get its input parameter.

Growth primitives are used to instantiate new cells and carry on the growth of artificial organism. Cells use data communications primitives to introduce computed data into the net and consequently interact with the environment. When a cell receives its specialization (by means of “run” or “call” packets) it is ready to perform its own job, but sometimes it may require further specialization to obtain a better identification (e.g. in the case of several cells with the same specialization). The cells use a small amount of internal memory for local temporary storage, therefore some primitives to read/write data are essential. Summing up, every time a virtual hardware functionality is required or some action related to bio-inspired model of computation has to be performed, a base primitive is used.

At the current stage one of the interpreters supported by BME accepts programs written in a language called “BME-while language” (BWL). It has been developed together with BME environment and supports all base primitives though it has a very simple grammar (there is only the “while-do” control flow statement) and it only manages with float data type. BWL defines a small set of arithmetic operators (+, *, /, -, div, mod) that, together with “while” statement, constitutes cellular operations.

5. ANALYSIS AND EXPERIMENTS

5.1 Complexity Analysis

In Figure 4 a simple BWL program for calculation of the following expression (*finite series*) is shown:

$$\sum_{j=0}^{l} f(x_j) \quad (1)$$

This text has been partially written in pseudo-code, because simple BWL syntax causes difficult readability.

We use the *finite series* process to describe our approach to computational analysis. As shown in figure 4-b, finite series process is intended as a complete binary tree where every node (root, internal nodes and leaves) is implemented by a BME-thread: internal nodes, that always have two offspring, and root performs a single sum, leaves perform a given $f(x_j)$. For such kind of binary tree, if the number of internal nodes is i, the number of leaves (i.e. the number of series term) is i+1. The growth (see figure 4-c) starts from the root (i.e. “Main” thread) and continues until all needed threads are instantiated. Leaves produce data and then expire. These data are caught by parents that produce their result and expire. This process goes on up to the root which prints the final result.

Spatial distribution allows to exploit parallelism: supposing that all interactions are parallel, a sum takes a single temporal unit, and function f takes a constant time $C(f)$, then complexity for calculation (see Figure 4-b) of a finite series is

$$t = \lambda\lceil \log_2(i+1)\rceil + (\lceil \log_2(i+1)\rceil - 2) + 2C(f) \quad (2)$$

where λ is the average interaction time between offspring and parents, i is the number of internal nodes, $\lceil \log_2(i+1)\rceil$ is the height of the tree, $\lceil \log_2(i+1)\rceil$-2 is the time used for parallel sums for every tree level and $2C(f)$ is the time consumed for the last two tree levels (where f is computed). In the simpler case:

$$i = 2^1 - 1, 2^2 - 1, 2^3 - 1, ..., 2^n - 1, ...$$

f is computed only in the last level nodes and computation time is

$$t = (\lambda + 1)(\lceil \log_2(i+1)\rceil) + C(f) \quad (3)$$

Nevertheless, in both cases, finite series BWL algorithm has a $\boldsymbol{O}(\log(l))$ complexity with a resources spatial occupancy of $2i$+1 cells.

The same algorithm can be mapped on a k-ary tree structure: all internal nodes have to generate k offspring and perform k-1 sum of caught results. For a k-ary tree, if number of internal nodes is i, then the number of leaves is:

$$(k-1)i + 1 \quad (4)$$

Consequently a process for a series of l terms has a number of internal nodes equal to

$$\left\lceil \frac{l-1}{k-1} \right\rceil \quad (5)$$

and can be mapped on a surface with a quantity of nodes equal to

$$\left\lceil \frac{l-1}{k-1} \right\rceil + (k-1)\cdot\left\lceil \frac{l-1}{k-1} \right\rceil + 1 = k\left\lceil \frac{l-1}{k-1} \right\rceil + 1 \quad (6)$$

and with an amount of unused resources equal to

$$(k-1)\cdot\left\lceil \frac{l-1}{k-1} \right\rceil + 1 - l \quad (7)$$

Eventually, equations (2) and (3), can be easily generalized:

$$t = \lambda(k-1)\left\lceil \log_k\left((k-1)\cdot\left\lceil \frac{l-1}{k-1} \right\rceil + 1\right)\right\rceil + \left(\left\lceil \log_k\left((k-1)\cdot\left\lceil \frac{l-1}{k-1} \right\rceil + 1\right)\right\rceil - 2\right) + 2C(f) \quad (8)$$

$$t = (\lambda+1)(k-1)\left\lceil \log_k\left((k-1)\cdot\left\lceil \frac{l-1}{k-1} \right\rceil + 1\right)\right\rceil + C(f) \quad (9)$$

If $k = l$ then the tree has a root with l leaves and, from (9) it easy to show that complexity is $\boldsymbol{O}(l)$.

The finite series example shows that an algorithm can be mapped in different ways and performances change in function of the distribution degree. The finite series example highlights the issue about *trade-off between distribution of computation and grouping operation inside nodes*: keeping k small, application is highly distributed with simple threads; increasing k, distribution decreases and operations are concentrated inside the nodes. Anyway, the complexity analysis doesn’t comprehend a real investigation about interactions influence on performances. Communication is just represented as a single parameter (λ), but nothing is said about its behavior (e.g. whether it changes in function of mapping rule or cell interconnection degree); it is considered as a constant hidden by $\boldsymbol{O}$ notation.

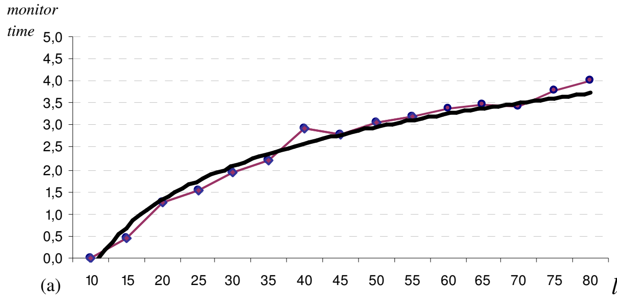
Generally speaking, every application has its own internal logic that leads mapping on the grid; at the same time, threads disposition on the space, lattice physical properties (i.e. connectivity, dimensions etc.) and routing rules affect internal communications of the application so deeply that they are very difficult to predict. With no evidence it is difficult to achieve, by means of a theoretical analysis, information about the right trade-off between *distribution* and *grouping operations inside nodes* and it is not possible to evaluate scalability of computation.

5.2 Experimental analysis

BME provides a useful simulation environment for application testing and performances evaluation; the results obtained by a simulation take into consideration how distribution affects communication, hence they are useful to achieve information about right trade-off between distribution and grouping operations inside functional nodes.

For our experiments we have used a 12x12 lattice in which every cell has four red-cells (as in Figure 1-a on the top) and a non deterministic routing rule that grants a uniform distribution in all directions (see Figure 4-c) and relatively small variation of value of λ against the number of allocated resources. A built-in monitoring tool is used to check the lattice: it periodically scans the status of computation and returns an estimation of time equal to the inverse of concurrency degree, since all concurrent activities would be performed simultaneously in a real system. The sum of all returned values is an index related to application execution time. We use a dynamic sampling technique which consists in thinning out monitor scanning activity when the amount of active threads decreases, and in intensifying it when this amount increases:

$$\Delta t_{i+1} = -\beta \cdot \log_\rho\left(1 + \frac{1}{\alpha} - \frac{1}{c_i + r_i + 1}\right), \quad \alpha > 1, \rho > 1 \quad (10)$$



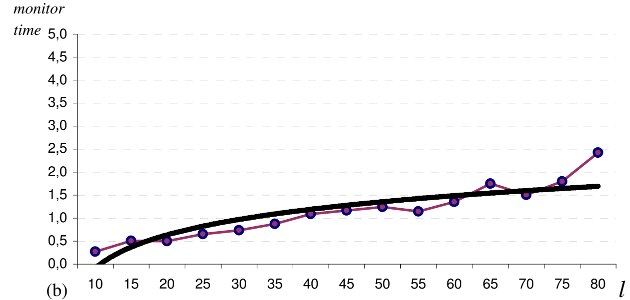
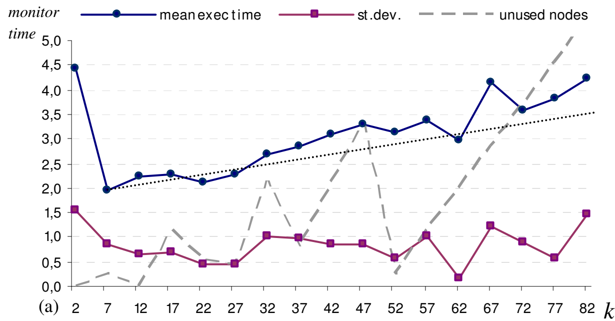


Figure 5. Experimental values and best fit curve for a binary tree shaped finites series of *l* elements confirm theoretical logarithmic behaviour. (a) A single red-Cell interaction takes ~0.05 ms and a cell executes the entire code in ~100 ms. (b) A single red-Cell interaction takes ~0.05 ms and a cell executes the entire code in ~0.4 ms.



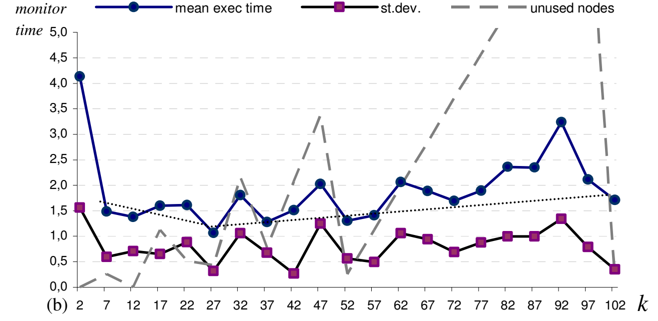


Figure 6. Average values of the execution time for finites series process (a hundred elements) based on *k*-ary trees and related standard deviation. The grey dotted line shows local minimum behavior and the dashed line shows the amount of unused nodes for a *k*-ary tree. (a) A single red-Cell interaction takes ~0.05 ms and a cell executes the entire code in ~100 ms. (b) A single red-cell interaction takes ~0.05 ms and a cell executes the entire code in ~0.4 ms.

When concurrency degree is low, computation advances too quickly to obtain a precise time estimation: the fewer threads are running, the higher measure error is. Therefore, having a higher sampling frequency when concurrency is intense, the weight of incorrect values collected by the monitor during low concurrency phases is less significant.

Equation (10) defines the sampling time at the step $i+1$ as a function of concurrency degree at step i in order to adapt sampling activity to computation. Parameter β is the base amplitude of sampling interval, parameter α settles the sampling time variation amplitude (higher α values imply a wider interval) and ρ sets the speed of variation (higher ρ values imply a higher speed).

The simulator allows user to set cellular operations and the net communication rate, but our monitoring tool returns a value in terms of number of samplings that is dependent just on concurrency degree and some parameters. The returned behavior of an application, loaded with different cellular and/or net velocity, will have a different degree of precision, because the monitor sampling rate cannot be increased over and over; hence slower communication/cellular operation rate allows a better precision of results, but faster rate allows testing more complex applications in shorter time.

We have mapped a binary tree shaped finite series with different values of l in order to settle parameters of our monitor and achieve a good level of precision: we have obtained the results shown in Figure 5 by using two different settings for the cellular operation rate. The binary tree process is suitable for monitor calibration, because the amount of instantiated threads only changes, while cells specialization and interactions are constant against l.

After monitor calibration we have executed different processes for calculating a finite series of a hundred elements using trees characterized by different values of *k.* Variations on tree offspring degree (*k*) makes threads more time-consuming (i.e. more sums have to be performed inside nodes) and have great impact on λ behavior, since the distance between offspring and parents increases against *k*: Our aim is to understand, by means of simulations, what the best values for k are. Every process has been executed twenty times in order to evaluate the steadiness of monitor responses: in Figure 6 are represented the average values of processes execution times and the corresponding standard deviation for every tested value of *k* using two different settings for cellular operation rate.

In the first case (figure 6-a) communication time is much lower than cellular execution time; for low values of *k*, a too high communications overhead, caused by long distances among interacting elements, makes the advantages given by parallelism useless, while between *k*=7 and *k*=22 parallelism is well exploited. Increasing *k* more and more, functional node "inertia" negatively affects computation. In the second case (figure 6-b) communication time and cellular execution time are almost equal[5]; the optimum interval is centered on *k*=27, the modification on cellular operation rate has changed the result in an expected way, in fact if cells are faster, to place more operations inside them is profitable. Moreover, "routing distortions" have great influence on results: when there is a high number of unused nodes (see equation 7), messages paths are on average longer, and this

[5]In general cellular message passing involve more than one hop.

effect is more evident when red-cell interaction latency is comparable to cellular operations rate. In both cases standard deviation is always upper-bounded and tends to decrease around local minimum of execution time; therefore the monitor estimations have a good reliability.

Parallelism is not an easily manageable resource: in the case of finite series with a very low communication latency, a too high or too low distribution may lead to a loss of performance, but, in general, functions representing the trade-off between spatial distribution and the grouping of operations inside nodes could be very complex with several local minimums; such trade-off could depend on more than one parameter, it is affected by messages routing and, as we have shown, it is strongly related to the ratio between functional units and communication units rate.

6. TOWARDS EVOLUTION

BME provides an environment with precise properties and laws. Artificial organisms rise and grow in this environment, which affects their structure, but not their genome. Code of organisms directly influences behavior, spatial resources allocation and parallelism degree: a different code means a different application mapping. Since variation on code are potentially infinite, it may be useful to introduce some evolutionary technique [25] with the aim of creating instances of computational entities which evolve towards better performances without explicit user interventions.

It is relatively easy to make a finite series process evolvable: the idea is to map a population of organisms with different distribution degrees on the lattice (i.e. different values of k stand for different application code), and employ an evolutionary algorithm in order to improve population, generation by generation.

The implemented evolutionary algorithm operates on a randomly generated initial population with k comprised between 2 and 82, applying the principle of survival of the fittest. At each generation, fitness (the inverse of execution time) is computed for every individuals and a new set of parents is created by roulette wheel method; selected individuals are bred together using intermediate recombination (see fig. 7) and reinserted by replacing a subset of parents uniformly at random.

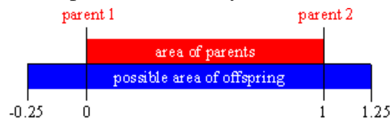


Figure 7. Area for variable value of offspring compared to parents in intermediate recombination.

Figure 8 represents the average k for every generation up to the hundredth: smaller values stand for a greater average distribution degree. Every population consists of 100 individuals; for each step of the algorithm, 50 parents are selected and 25 offspring (which will randomly replace 25 parents in the next generation) are generated. Figure 8 shows a gradual approach to the optimal values found in the previous section, when we have used cellular operation rate much slower than red-cell communication rate (see Fig. 6-a); moreover standard deviation related to values of k in each generation decreases generation by generation.

At the present stage a single cell is programmed to execute evolutionary algorithm, i.e. it collects results, evaluates fitness and breeds a next generation. This is a raw method used to explore the behavior of evolutionary approach applied to BME-organisms.

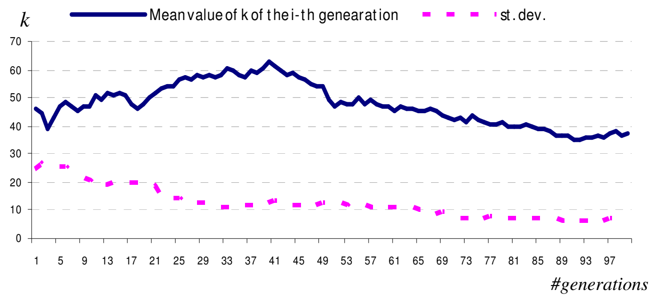


Figure 8. Mean values of *k* for every generation produced by evolutionary algorithm applied to finite series process and relative standard deviation.

The real aim is to enrich bio-inspired model of computation improving artificial organism specification (i.e. genome) by providing a reproductive capability. Given a pair (or a set) of organisms, they could be able to create a new organism for the same application, by using services or operations derived from parents. Every organism should have a finite life time according to its own fitness, so that every organism has a higher probability to reproduce itself as long as its own fitness is higher.

A similar approach could be exploited for problem solving (see [23] and [24]). If this "functional merging" among individuals involves application logic, offspring could evolve towards different applications. In this case, the evolutionary trend should be dictated by different definition of fitness, i.e. fitness doesn't have to be given as a function of execution performances, but according to the correctness of results.

7. CONCLUSIONS AND FUTURE WORKS

BME provides a good substrate to analyze different solutions for a highly dynamic spatial computation. By means of simulator, we want to propose empirical analysis and evolutionary techniques as an approach to evaluate performances and trade-offs among different choices about resources allocations, and communications. This kind of analysis could provide interesting results about new models of computation suitable for machines based on molecular scale devices, where the huge amount of resources will upset conventional architectural and computational paradigms (see [1]-[4]).

Application of *evolutionary processes* to complex applications, both for performances and problem solving is not an easy challenge: it will require a much higher number of cells, and thus it will impose to exploit resources of more than one physical machine, furthermore it will need a more precise genome definition e.g. by means of "meta-genes" to distinguish functional classes of cellular code subsets. At the current stage we are developing a distributed version of BME: the basic idea is to exploit a computer network in which every machine executes an instance of the simulator.

Artificial organisms have a dynamic structure in the sense that cellular allocation is subjected to computation. BME offers a good base to highly improve dynamism and to confer properties of *adaptation* to organisms. It could be possible to allow cells to keep traces of their own interactions, and move towards other positions to minimize packets time arrival. This behavior would make artificial organisms able to analyze themselves and to improve their structure, taking a shape more suitable to environment laws (i.e. routing rules and interconnection topology) and conditions.

Fault-tolerance policies impact has not been analyzed yet. At the molecular scale the computational devices will be less reliable than actual silicon based ones (see [1],[2] and [4]), so a native run-time recovery capability for local (i.e. cellular) computational state is very important to obtain a reliable computational machine. BME now provides a set of uniformly distributed spare cells (i.e. staminal cells), but they are not used yet and fault-tolerant policies for run-time computation recovery are currently a work in progress.

8. ACKNOWLEDGMENTS

This work was supported by “Silvio Tronchetti Provera” Foundation. The authors would like to thank AST labs (ST-Microelectronic) team for support and discussions.

9. REFERENCES

[1] Lisa J K Durbeck, Nicholas J Macias. The Cell Matrix: an architecture for nanocomputing. Nanotechnology, no. 12, pp. 217-230, Sep. 2001.

[2] Paul Beckett, and Andrew Jennings. Towards nanocomputer architecture. In Australian Computer Science Communications, Proceedings of the seventh Asia-Pacific conference on Computer systems architecture, Vol.6, Jan. 2002.

[3] Paul Beckett. VLSI in the nanometer era: Exploiting multiple functionality for nano-scale reconfigurable systems. In Proceedings of the 13th ACM Great Lakes symposium on VLSI, Apr. 2003.

[4] Ferdinand Peper, Member, Jia Lee, Fukutaro Abo, Teijiro Isokawa, Member, Nobuyuki Matsui and Shinro Mashiko. Fault-Tolerance in Nanocomputers: A Cellular Array Approach. IEEE Trans. on nanotechnology, vol. 3, no. 1, pp. 187-201, March 2004.

[5] Walid A. Najjar,Wim Böhm Bruce A. Draper, Jeff Hammes Robert Rinker, J. Ross Beveridge, Monica Chawathe and Charles Ross. High-Level Language Abstraction for Reconfigurable Computing. IEEE Computer - Innovative Technology for Computing professionals, pp. 63-69, Aug. 2003 .

[6] J.P.Hammes, R. E. Rinker, D. M. McClure, A. P. W. Böhm, W. A. Najjar. The SA-C Compiler Dataflow Descritpion. ACM Transactions on Embedded Computing Systems (TECS) Vol. 2, Issue 4, pp. 560-589, Nov. 2003.

[7] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea and Seth Copen Goldstein. Spatial Computation. In Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2004, Boston, Massachusetts, USA.

[8] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea and Seth Copen Goldstein. C to Asynchronous Dataflow Circuits: An End-to-End Toolflow. In Thirteenth International Workshop on Logic and Synthesis, June 2004, Temecula, California, USA.

[9] Seth Copen Goldstein and Dan Rosewater. What Makes a Good Molecular Scale Computer Device?. Technical Report CMU-CS-02-181, September 2002.

[10] Sven Bodo Scholz. Single Assignment C -- Functional Programming Using Imperative Style. In Functional Languages Implementation Workshop. Paper 21.

[11] Luca Benini, Giovanni De Micheli. Network On Chip: A New SoC Paradigm. In Design, Automation and Test in Europe Conference and Exhibition, Paris, France, March 2002.

[12] Girish V. Varatkar, Radu Marculescu. On-Chip Traffic Modeling and Synthesis for MPEG-2 Video Applications. IEEE Trans. on Very Large Scale Integration systems, vol. 12, no. 1, pp. 108-119,Jan. 2004.

[13] C. Teuscher, D. Mange, A. Stauer, and G. Tempesti. Bio-Inspired Computing Tissues: Towards Machines that Evolve, Grow, and Learn. In Fourth International Workshop on Information Processing in Cells and Tissues, Inter-University Micro Electronics Center Kapeldreef 75 3001 Heverlee (Leuven) Belgium, Apr. 24, 2001.

[14] Gianluca Tempesti, Daniel Roggen, Eduardo Sanchez, Yann Thoma, Richard Canham and Andy Tyrrell. Ontogenetic Development and Fault Tolerance in the POEtic Tissue. In The 5th International Conference on Evolvable Systems: From Biology to Hardware, March 2003.

[15] Yann Thoma, Eduardo Sanchez, Juan-Manuel Moreno Arostegui and Gianluca Tempesti. A Dynamic Routing Algorithm for a Bio-Inspired Reconfigurable Circuit. Field-Programmable Logic and Applications pp. 681-690, 2003.

[16] Andrè DeHon. Compact, Multilayer Layout for Butterfly FatTree. In Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 206--215, July 9-12, 2000.

[17] C. S. Lent and P. D. Tougaw. A device architecture for computing with quantum dots. Proc. IEEE, vol. 85, pp. 541–557, 1997.

[18] Craig S. Lent, P. Douglas Tougaw, Wolfgang Porod, and Gary H. Bernstein. Quantum Cellular Automata. Nanotechnology 4, pp. 49-57 ,1993.

[19] P. Douglas Tougaw and Craig S. Lent. Dynamic behavior of quantum cellular automata. Journal of Applied Physics, Vol. 80, pp. 4722-4736 Oct. 15, 1996.

[20] Konrad Walus, Timothy J. Dysart, Graham A. Jullien and R. Arief Budiman. QCA Designer: A Rapid Design and Simulation Tool for Quantum-Dot Cellular Automata. IEEE Trans. on nanotechnology, Vol. 3, no. 1, March 2004.

[21] C. Ghezzi, Mehdi Jazayeri, Programming Language Concepts. Third Edition, Wiley 1998.

[22] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In ACM SIGPLAN Notices, Vol. 29, pp. 31–37, Dec.1994.

[23] John R. Koza. Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical Report: CS-TR-90-1314, June 1990.

[24] John R. Koza and James P. Rice. Genetic Generation of Both the Weights and Architecture for a Neural Network. In International Joint Conference on Neural Networks, Vol. 2 pp. 307-404, Seattle, Washington, USA, 1991.

[25] Melanie Mitchell. An introduction to genetic algorithms. MIT Press, 1996.

[26] Peter Bentley. Aspects of Evolutionary Design by Computers. In Proceedings of the 3rd On-line World Conference on Soft Computing in Engineering Design and Manufacturing, 1998.