

Making Cortically-Inspired Sensorimotor Control Realistic for Robotics: Design of an Extended Parallel Cellular Programming Model

Olivier Ménard, Stéphane Vialle, Hervé Frezza-Buet

▶ **To cite this version:**

Olivier Ménard, Stéphane Vialle, Hervé Frezza-Buet. Making Cortically-Inspired Sensorimotor Control Realistic for Robotics: Design of an Extended Parallel Cellular Programming Model. International Conference on Advances in Intelligent Systems - Theory and Applications - AISTA 2004, Nov 2004, Luxembourg, 6 p. inria-00099899

HAL Id: inria-00099899

https://inria.hal.science/inria-00099899

Submitted on 26 Sep 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Making Cortically-Inspired Sensorimotor Control Realistic for Robotics: Design of an Extended Parallel Cellular Programming Model

Olivier Ménard
Loria[1], Supélec[2]
[1] Bât Loria, Campus Scientifique,
BP239, F-54506 Vandœuvre-lès-Nancy
Email: Olivier.Menard@supelec.fr

Stéphane Vialle
Supélec[2]
[2] 2, rue Edouard Belin
F-57070 Metz Cedex
Email: Stephane.Vialle@supelec.fr

Hervé Frezza-Buet
Supélec[2]
[2] 2, rue Edouard Belin
F-57070 Metz Cedex
Email: Herve.Frezza-Buet@supelec.fr

***Abstract*— This paper introduces a multi-layer software architecture, that allows smart design and implementation of complex cortical neural networks, and efficient parallel execution on multiprocessor machines. The developer implements cortical networks sticking to their natural fine grained formalism, without caring about its mapping on coarse grained parallel computers. The developer can use a set of graphical tools to easily develop and debug the cortical systems.**

Some experiments of new cortical model design on multiprocessor PCs are introduced, and some performance measurements are given. This software suite is operational and currently in use in our laboratory to control a robotic am through cortical neural networks running on multiprocessor PCs.

I. Introduction

The artificial neural network approach in the field of Computer Science supports the idea that some smart computation can be designed on the basis of a set of fine grain processing units, the artificial neurons, that are connected to exchange information frequently. This kind of computational paradigms seems to lead "naturally" to parallelism, since it is clearly inspired from the nervous system of animals, where neurons actually form huge interconnected networks of processing units that all run continuously.

Of course, when brought from biology to Computer Science, neural computation has had to be simplified, leading to many models of computation. In most cases, these simplifications led to some algorithms that have lost their intrinsically fine grained and highly parallel nature. This is the case for classical multi-layer perceptron [1], where computation of units has to be centralized to feed the network forward successive layers, and then to feed it back with error gradient in the reverse order. Another famous case is the Self Organizing Maps by Kohonen [2], where a maximum matching unit has to be found in a sequential way. Moreover, the way such neural network algorithms can be implemented on modern parallel architectures is quite far from their fine grained parallelism. Naive programmers expect to deal at the level of the neural units, corresponding to fine grained computation with a lot of elementary communications. But to run efficiently, modern parallel machines need coarse grained programming, with a limited number of significant processes exchanging long messages [3].

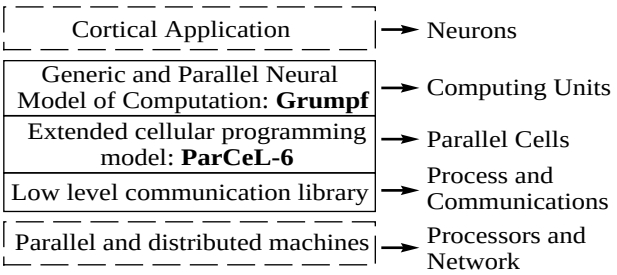


Fig. 1. Software architecture designed to easily define, implement and run cortical systems on parallel architectures

In this paper, a multilevel architecture is presented (see figure 1), that reconciles artificial neural networks to the intrinsic parallelism one can expect from them and to execution on modern and general purpose parallel computers. A high level *generic and parallel neural model of computation* allows a smart programming based on numerous computing units. A medium level *extended cellular programming model* allows to map these numerous computing units on some powerful processors of modern parallel machines. Thus, running large numbers of neurons becomes feasible, without any effort from the neural network designer, allowing to address some complex tasks as multi-joint robotic arm control, when no model of the device is known.

The general neural network model is first discussed in next section, with its limitations. Then, the way it is made parallel on shared memory computers and on clusters is presented. Last, the performance of the model is tested on a cortically inspired architecture for sensori-motor control.

II. A generic and parallel neural model of computation

We have designed a computational model for artificial neural networks that can be formalized as follows. This model

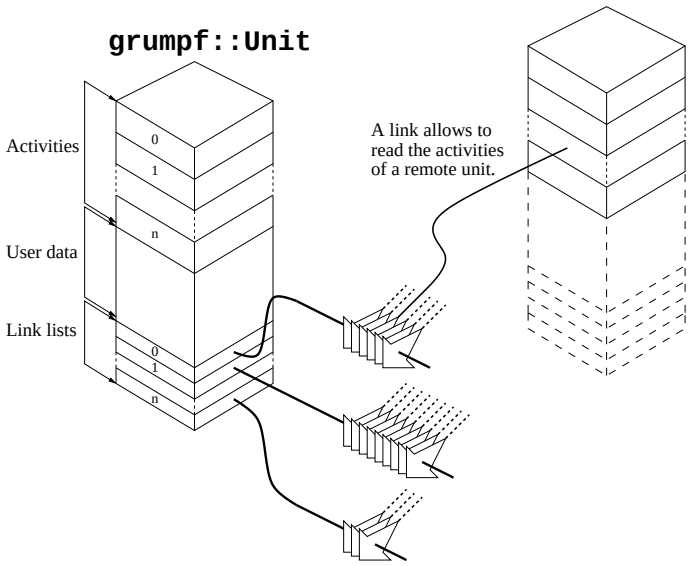


Fig. 2. Units and links in the `grumpf` formal model. A link is a unidirectional read only access to the activities of some remote unit. For convenience, links are put in lists.

is available [4] as a `C++` library, named `grumpf` (`gr`aphical `u`tilities for the `m`odeling of `p`arallel `f`unctions), that provides classes to implement a neural network in that context, and provides graphical tools for viewing and debugging (see figure 3). Let us mention here that the `grumpf` architecture allows the design of neural models that can run on a remote parallel machine, but that can be controlled and viewed by TCP clients on a development PC. This property is hidden to the programmer.

The formal model consists of a set of computational units, connected together. Each unit allows the computation of a fixed number of floating point values, its activities. Units can have various number of activities in that set, but for a given unit, the number of activities is kept constant.

Connection between units are made by the use of links. A link is an unidirectional reading access, owned by a unit to read activities of another unit (see fig. 2). This is closer to biology than bidirectional links that are often used by classical neural techniques, since real neurons provide their output on their axon. From the axon, this output is read by synapses of the dendritic tree of some other neurons. Last, for programming convenience, links owned by a unit can be put in separate lists, so that the unit can easy compute, for example, the mean of the values got from links in list number 1, the max of some other values got from links in list number 2, and so on.

Units in the model update their activities periodically. Updating *all* the units *once* in the model defines a computation step, and the model behavior is the result of successive computational steps. Two updating policies are defined by the model. The first one is *synchronous* updating (or *full backup* update). This consists of taking all units in the model, and compute the modification of their activities in some hidden buffer, so that no modification really occurs at the level of each unit. When all updates have been performed, before starting next computational step, the stored modifications that have been saved in the hidden buffer are applied to the unit. Then the units actually change their activities. With this policy, as one computation step consists of evaluating all the units once, the order of evaluation is not meaningful. The second kind of updating policy is the *asynchronous* one. At each computational step, all the units are updated in *random* order, and updating a unit immediately changes the state of its activities.

These two kinds of updating policies are crucial in the formal model, since they may lead to different results for a given neural architecture. The former one, that is the synchronous updating policy, is suitable for process like the famous game of life by Conway, and is not that realistic when the `grumpf` computational model is used for the design of biologically inspired models, since neurons are not intrinsically synchronized[1]. The latter updating policy, the asynchronous one, is rather suitable for relaxation processes, as the one proposed in the model of content addressable memory by Hopfield for example [5]. Asynchronous updating policy is the one that is actually used in the robotic application mentionned in this paper to evaluate performances of the computational model.

To sum up, designing a neural architecture with such a formal model, and in practical cases with the available `grumpf` library, consists of designing first some kinds of unit, each one having its specific updating process, then creating numerous instances of units of these defined kinds, and last putting connections between them. Once this set up is performed, the cycles of computational steps have to be started. Then, the behavior of the neural network is strictly the emerging result of the local neural computations. With such an approach, it is clear that implementing a multi-layer perceptron, or even a Kohonen map is not easy, requiring computational tricks, but the very purpose of the model is to help designers that want to preserve the parallel and unidirectional nature of real neurons in their models. This is the case for the model of sensori-motor control used in this paper for illustration.

Last, let us mention that the `grumpf` library allows the neural network designer to express his or her model as a fine grained parallel system, and the efficient parallelization of that system on modern general purpose parallel computers is not in charge of the designer. To allow such an efficient automatic binding between neural models and computer hardware architectures, `grumpf` is grounded on an extended cellular automata model (`ParCeL-6`) that is described now.

III. An extended cellular programming model

As detailed in previous section, `grumpf` hides to the developer the difficulty to map fine grained units on coarse grained parallel architectures, by the use of a parallel model of cellular automata-based computation. Some generic parallel libraries (like `P-Threads` or `MPI`) have existed from many

[1]This is not in contradiction with the fact that some emerging synchronism can be observed within a neural population, but this discussion is out of the scope of this paper.

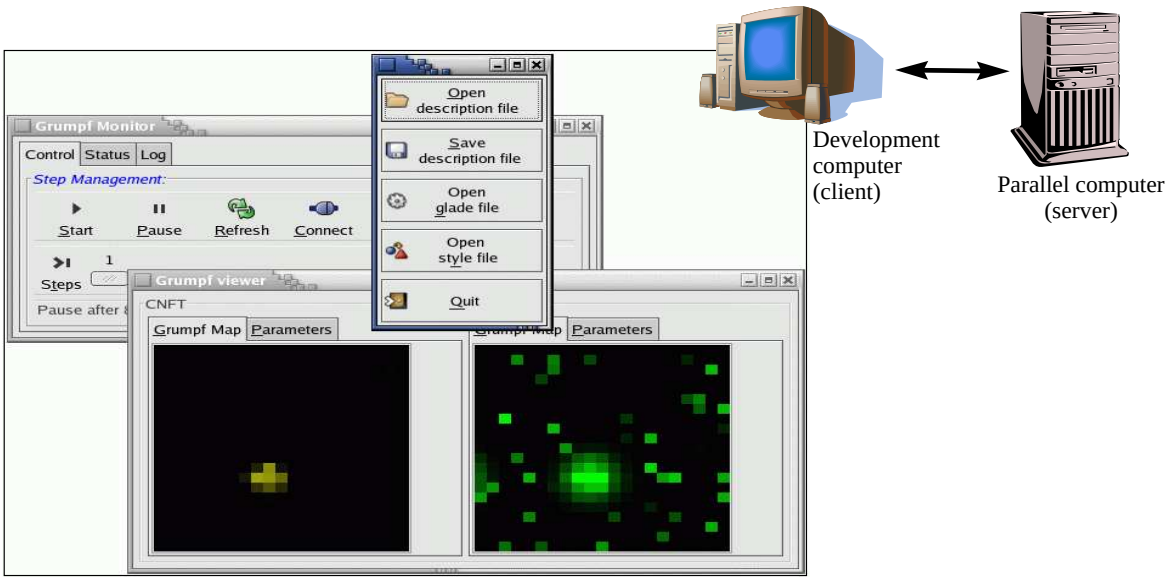


Fig. 3. Highly customizable graphical clients allow the viewing and the debugging at runtime of the neural `grumpf` parallel process running on a computing server

years, but they are more suitable to implement medium and coarse grained parallel algorithms. At the opposite, some parallel libraries of cellular automata are available to implement fine grained applications on modern parallel architectures (Carpet[6],CDL[7]), but they appear too restrictive for our applications. For example, our actual cortical systems need to connect distant cells, not just neighbor ones.

So we have designed a new cellular and parallel programming model, that is implemented by the `ParCeL-6` library, presented in this section.

A. A cellular network interacting with a sequential program

Previous ParCeL projects have shown a pure cellular model is hard to program. So, a `ParCeL-6` program is composed of a sequential and classical program (the main program) that initializes the management of a cellular network, creates at less a part of this network, calls some cellular net routines (usually in a computation loop), and finally removes this cellular network and its data structures.

At the beginning of a `ParCeL-6` program no cell exist. The main program has to create the first cells before to run the first cellular computing cycle. Then new cells can be created or some cells can be removed by already existing cells or from the main program at next inter-cycles. But in many applications all the cells are created from the main program before the first cellular computing cycle starts and are never removed. When a cell is created a host processor is pointed out (with respect to statistic load balancing), and a unique registration number is affected to this cell. This number allows to identify the cell in all the cellular network, and the cell is created directly on its host processor and will stay on it.

B. A cyclic cell activation

A `ParCeL-6` cellular network has a cyclic running: the main program frequently calls a routine that runs one cellular computing cycle, then each cell is activated only once.

Inside a cell computing cycle, all cells are conceptually activated in parallel, and the cycle ends when the longest cell finishes its computation. However, when implemented on modern and general purpose computers and processors, many cells are managed per processor and cells on one processor will be run sequentially. To clarify this situation, ParCeL-6 programming model specifies that:

- no assumption can be done on the order of cell executions on a processor,
- it is possible for the application programmer to change this cell execution order, to check if it impacts on the result of its program, but he or she can not specify the exact order of cell executions.

C. Cell composition

Each `ParCeL-6` cell is a kind of virtual light-process, that has a private memory space. It is composed of:

- some private variables, that can be scalar variables, or arrays, or dynamic and extensible data structures,
- some pointers on executable code (5 functions, see further) to run when the cell is activated,
- some parameters (read only) set by the main program when the cell is created, and that can be updated at each inter-cycles by the main program,
- an output: an array of ”double”, with a size fixed at the cell creation,
- some input channels (some private variables with a specific data type), that can be created dynamically, and that

need to be explicitly connected to the output of other cells,
- a registration number, fixed at the cell creation, that identifies the cell in the entire cell network.

D. Functions controlling the cell behavior

The behavior of a cell is defined by three routines: one for its first activation cycle (`init` function), one for its last activation cycle (`term` function), and one for all its other activation cycles (`iter` function). All these functions are fixed at the cell creation. Usually, the `init` cell function is used to allocate and initialize some cell private variables, and to connect the input channels of the cell to some cell outputs. The `term` function is basically used to free some private variables, disconnect the input channels, and sometimes to save some data on disk.

Two other functions allows a cell to react on death or birth of other cells, at the end of each computation cycle. These functionalities concern highly dynamic cellular systems and have not been used yet.

E. Request commands for cellular network management

Some commands exist to manage the cell network topology: to create and kill cells, and to connect (and disconnect) the input channel of a cell to the output of another cell. These commands are *requests* and their execution is split into two parts: a first small part that ends very quickly and mainly stores a definition of the job to do, and a second part that really does the job at the end of the cellular computing cycle. This second part is usually time consuming and needs some communications between processors. Finally, at the next computing cycle all *requests* have been entirely executed and the cellular network has been updated.

If some requests are impossible to execute, such as kill a cell already killed, or connect to a dead cell, then the request fails, but in any cases the cell network status remains coherent. For example, if several requests are emitted to kill the same cell, its `term` function is executed only once.

F. Cell communications

Communications between the sequential main program and the cellular network happen at the beginning and at the end of any cellular computation cycle. When the cellular network is inactive, the main program can send new parameters to the cells: global parameters for all cells (like camera images that would be the inputs of a neural network) or a set of specific parameters (different parameters for each cell). Moreover, at any inter-cycle the main program is able to collect individual results from each cell in a global result data structure.

Communications between cells go through *communication channels* that have to be explicitly established in the cell behavior functions. These connections are unidirectional and can evolve dynamically during the `ParCeL-6` program execution: new input channels can be created and connected, or existing ones can be re-connected to other cell outputs. To connect one of its input to the output of another cell, a reader cell has just to know the registration number of the writer cell. This registration number is passed to the reader cell into one of its parameters (usually by the main program).

Finally, the main program, or some ”mother cells”, create a lot of cells, send cell registration numbers to each cell into its parameters, and run the cell computation. Then each cell establishes its own connections during its `init` step, and starts its computation. This cell network construction has appeared a comfortable solution during our past experiments with a previous `ParCeL` project [8].

G. Inter-cell communication protocols

It exists three kinds of cell output in `ParCeL-6` model (*direct*, *buffered* and *hybrid* outputs), and an explicit command to refresh an input channel (`p6refresh(...)`). This leads to three different communication protocols between cells.

1) Direct communications: An input channel connected to a *direct* cell output will update its value immediately after a refresh command, but at most one time per cycle. Any other refresh command on the same input channel during the same cycle will be ignored, but other input channels of this cell, or input channels of other cells can be refreshed. This mechanism ensure a *direct* access to the connected output but only one time per cycle and per input channel. Note that if several input channels are connected to the same output, each input channel will accept a refresh command at the current cycle, and then different input channels can read different values.

The *direct* outputs and their refreshing mode lead to a kind of *asynchronous* running: cells are activated in parallel, they exchange data during the computation cycles, and the exchanged values depend on the different computation times they need to update their outputs. So, the program result can change with the number of processors and with the parallel architecture used.

2) Buffered communications: An input channel connected to a *buffered* cell output will be updated automatically but only between cellular computing cycles (at inter-cycles). Explicit refresh commands will be ignored. So, during a computing cycle all input channel values are fixed, and all cells connected to the same cell output read exactly the same value. So, the order of the cell computations has no impact on the communications, and the program execution becomes totally determinist, not depending on the number of processors.

The *buffered* outputs and their refreshing mode lead to a kind of *synchronous* running (*full backup* update).

3) Hybrid communications: An input channel connected to an *hybrid* cell output will have an intermediate refreshing mechanism: Not so *on demand* than a connection to a *direct* output, not so rare than a connection to a *buffered* output.

Hybrid cell outputs lead to a compromise between asynchronous and synchronous executions. Some experiments on relaxation problems and on cortical systems have shown the asynchronous mode leads to fast convergence of cellular networks (less computing cycles are needed), but to long execution times because *on demand* refresh generate unplanned communications that disturb the computations. At

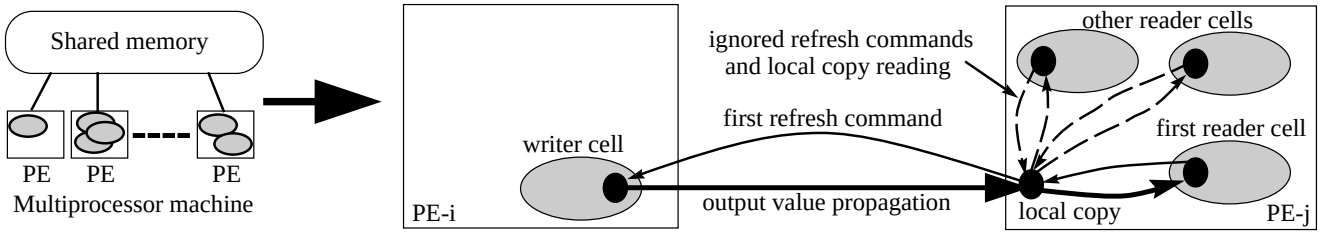


Fig. 4. Example of *hybrid* cell communications mechanism of `ParCeL-6.1` on shared memory parallel machine. Several cells on a same processor read the *hybrid* output channel of a cell located on another processor, and access to the same value: a local copy get by the first refresh command.

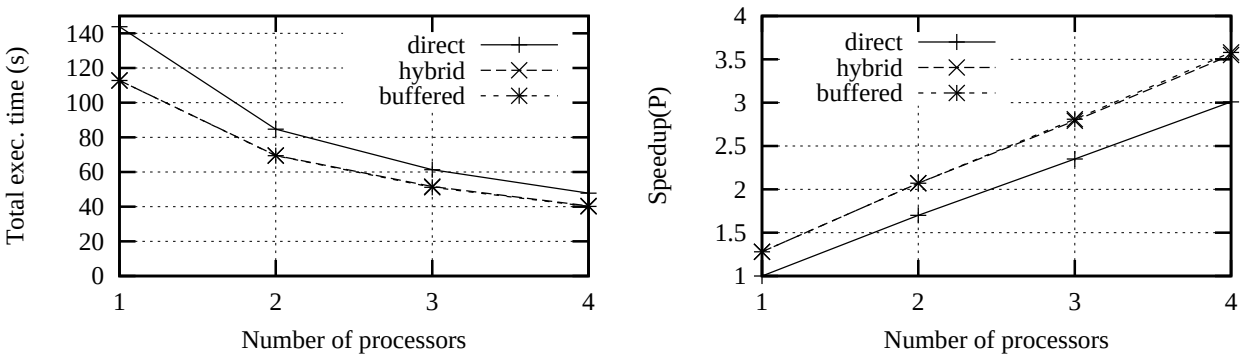


Fig. 5. Execution time and speedup *vs* sequential execution in *direct* mode, of a cortical control application on a 4-processor PC

the opposite, synchronous mode leads to slower convergence (it needs more computing cycles) but runs each cycle faster. *Hybrid* communication mode aims to converge fast and run fast. But details of this communication mode are not specified and depend on the architecture used (shared memory multiprocessors, distributed memory clusters, grids, ...). We have currently designed two *hybrid* modes, one for shared memory machines and one for clusters (see further).

IV. Two derived `ParCeL-6` models and implementations

A. `ParCeL-6.1` *model for shared memory architectures*

Classical shared memory architectures (SMP) and some hardware distributed shared memory architectures (DSM) support efficiently the memory sharing paradigm, and allow to implement efficiently all features of `ParCeL-6` model. `ParCeL-6.1` is a derived programming model of `ParCeL-6` designed for these architectures, and respects all features of `ParCeL-6` model.

`ParCeL-6.1` defines an *hybrid* cellular communication mode that leads to less communications between processors but still includes some *on demand* refresh of input channels. An input channel value is updated only after an explicit refresh command (like in *direct* mode), but only one time per cycle and per processor (not per cell). When several input channels located on a same processor are connected to a same *hybrid* cell output, the first refresh command executed on one of these input channels will update all these input channels for the current cycle (the next refresh commands will be ignored), see figure 4. Usually any cell asks to refresh its input channels before to read them, so all cells of a processor that are connected to a same *hybrid* output channel will read the same value (according to the first refresh command executed). But different processors will execute their refresh commands at different time, and can get different values of a same output channel.

With a medium number of processors the different cells connected to a same *hybrid* cell output are statistically spread on different processors, and each processor hosts several of these cells. So, on a large number of processors this *hybrid* communication model still leads to an *asynchronous* running, but accesses less often the cell outputs than the *direct mode*, and less disturbs the computations.

B. `ParCeL-6.2` *model for distributed memory architectures*

Distributed memory architectures, like clusters and grid, are interesting because they can scale until hundreds or thousands of processors. But usually they do not support a memory sharing paradigm, and can not run efficiently all features of `ParCeL-6` programming model. For example, the *direct* cell communication mode would not reach good performances. So, a new derived programming model has been defined: `ParCeL-6.2`.

Hybrid communication mode is a critical feature of `ParCeL-6.2`, and is different of the *hybrid* mode of `ParCeL-6.1` in order to be adapted to distributed memory architectures. A prototype of `ParCeL-6.2` is running on cluster since July 2004. But its *hybrid* communication mode

is still under development, the current performances are not satisfying.

V. Experiments and performances

The ability of the ParCeL-6 and grumpf suite to be used in practical cases has been tested on a robotic application. The application consists of controlling an articulated robotic arm so that the hand reaches visual targets. The point on this experiment is that it relies on an original bijama model of cortically-inspired computation. Applying bijama to such realistic problems [9] leads to the definition of many elaborated computational cells. The smart and easy design of such a model is provided by the C++ classe of grumpf librairies and assorted graphical tools, but the feasibility of the neural approach is ensured by the automatic parallelism that is obtained from ParCeL-6.

A technical specificity of ParCeL-6.1 is to be implemented and optimized to run efficiently on cheap multiprocessor PCs. These machines have a poor memory sharing mechanism, that need optimized parallel implementations to get performances on irregular computations. Figures 5 shows some performance measurements of bijama cortical application run on a 4-processor PC (4xPIII-Xeon-700MHz, 2MB cache memory/processor). Execution time decreases for any kind of cell output used, and parallel runs always speedup. As expected *direct* mode is the slowest mode and *buffered* mode is the fastest, but the interesting result is the *hybrid* mode runs as fast as *buffered* mode while communications remain partially *on demand*. Moreover, when compared to the sequential run of the *direct* mode, the *buffered* and *hybrid* modes exhibit greater speedup than *direct* one, and speedup difference seems to increase. *Buffered* and *hybrid* modes run faster and scale better than *direct* mode.

But the very gain is that, whereas buffered mode changes the network behavior from the correct direct mode, hybrid mode improves execution speed, while keeping properties of the direct mode. The influence of the updating mode has been evaluated on the CNFT algorithm described in [10], and in our cortical model [9]. This implements a relaxation mechanism in a bidimensional neural fields, leading to a competition from on-center off-surround connexions within the neurons. The result, in direct mode, is a bubble of activation, as predicted by the theory. In buffered mode, the neural network goes into an infinite oscillating state and is unable to stabilize on bubbles. Using hybrid updating mode doesn't prevent from oscillating states, at least for few processors. In our experiments, this effect disappears when using eight ParCeL-6 process (2 per real processor of a 4-processor PC), since in that configuration the dynamics of bubbles is similar to the one predicted by the theory, as with direct mode.

VI. Conclusion and perspectives

Our multi-layer architecture for smart design and efficient parallel execution of cortical neural networks is operational on multiprocessor PCs. It is currently in use in our laboratory to design new cortical models and to control a robotic arm from a 2-Opteron and a 4-PentiumIV machines.

The graphical interface of grumpf is still under development, and new functionalities will appear soon to improve cortical network design and debug.

The cluster version of ParCeL-6 is developed in the framework of the Grid-eXplorer project (French ACI project), in collaboration with researchers from Loria laboratory. It uses the SSCRAP[11] communication library developed at Loria laboratory, and some experiments on a large grid of several hundred machines is planned for 2005.

These perspectives allow to consider larger sets of neurones has being conceivable. This meets the need for more elaborated cortical architectures, with many more neural modules, for realistic robotic applications.

Acknowledgment

This research is supported by the Mirrorbot and Robea projects, and by the Region-Lorraine (France). Authors want to thank Jacques Weidig and Anca Ghitescu for grumpf developments, Laurent Casse and Radu Kopetz for ParCeL-6 implementation and optimization, and Jens Gustedt and Mohamed Essaidi for preliminary discussion about ParCeL-6.2 programming model.

References

[1] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," in *Neurocomputing: Fundations of Research (1989)*, J. A. Anderson and E. Rosenfeld, Eds. The MIT Press, 1958, pp. 89–92.

[2] T. Kohonen, *Self-Organization and Associative Memory*, ser. Springer Series in Information Sciences. Springer-Verlag, 1989, vol. 8.

[3] Y. Boniface, F. Alexandre, and S. Vialle, "A bridge between two paradigms for parallelism: Neural networks and general purpose mimd computers," *IJCNN-99: International Join Conference on Neural Networks*, July 1999, washington DC, USA.

[4] Http://www.metz.supelec.fr/˜ersidp/Software/Grumpf/Root.html.

[5] J. Hertz, A. Krogh, and R. G. Palmer, *Introduction to the theory of neural computation.* Addison Wesley, 1991.

[6] D. Talia, "Solving problems on parallel computers by cellular programming," *Proc. of the 3rd Int. Workshop on Bio-Inspired Solutions to Parallel Processing Problems BioSP3-IPDPS, LNCS, Springer-Verlag*, pp. 595–603, May 2000, cancun, Mexico.

[7] K.-P. V. S. W. C. Hochberger, R. Hoffamnn, "Cellular processing environment," *PARELEC98*, 1998, bialystok, Poland.

[8] Y. Boniface, F. Alexandre, and S. Vialle, "A library to implement neural networks on MIMD machines." *LNCS vol. 1685, Proceedings of Euro-Par'99*, pp. 935–938, 1999, Toulouse, France.

[9] O. Ménard and H. Frezza-Buet, "Rewarded multi-modal neuronal self-organization: Example of the arm reaching movement," in *International Conference on Advances in Intelligent Systems - Theory and Applications*, 2004, Luxembourg.

[10] J. G. Taylor, "Neural networks for consciousness," vol. 10, no. 7, pp. 1207–1225, 1997.

[11] M. Essadi, I. G. Lassous, and J. Gustedt, "Sscrap: Soft synchronized computing in rounds for adequate parallelization," INRIA, Research report RR-5184, May 2004.