The parXXL Environment: Scalable Fine Grained Development for Large Coarse Grained Platforms

Jens Gustedt, Stéphane Vialle, Amelia de Vivo

▶ **To cite this version:**

Jens Gustedt, Stéphane Vialle, Amelia de Vivo. The parXXL Environment: Scalable Fine Grained Development for Large Coarse Grained Platforms. PARA-06: Worshop on state-of-the-art in scientific and parallel computing, Jun 2006, Umea, Sweden. pp.1094-1104, ⟨10.1007/978-3-540-75755-9_127⟩. ⟨hal-00280094⟩

HAL Id: hal-00280094

https://centralesupelec.hal.science/hal-00280094v1

Submitted on 25 Jun 2008

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.

HAL Authorization

The parXXL Environment: Scalable Fine Grained Development for Large Coarse Grained Platforms

Jens Gustedt[1], Stéphane Vialle[2], and Amelia De Vivo[3]⋆

[1]INRIA Lorraine & LORIA, France. Jens.Gustedt@loria.fr
[2]SUPELEC, France. Stephane.Vialle@supelec.fr
[3]Università degli Studi della Basilicata, Italy.

Abstract. We present a new integrated environment for cellular computing and other fine grained applications. It is based upon previous developments concerning cellular computing environments (the ParCeL family) and coarse grained algorithms (the SSCRAP toolbox). It is aimed to be portable and efficient, and at the same time to offer a comfortable abstraction for the developer of fine grained programs. A first campaign of benchmarks shows promising results on clusters.

1 Motivations and objectives

Nowadays, many research areas consider multi-scale simulations based on *ab initio* computations: they aim to simulate complex macroscopic systems at microscopic level, using fundamental physical laws. Obviously, huge amounts of CPU are mandatory to run these simulations. Modern supercomputers and Grids, with large and scalable number of powerful processors, are interesting architectures to support these simulations.

However designers and developers of algorithms and code for large scale applications are often confronted with a paradoxical situation: their modeling and thinking is *fine-grained*, speaking *e.g.* of atoms, cells, items, protein bases and alike, whereas modern computing architectures are *coarse-grained* providing few processors (up to several thousands $\approx 10^3$) to potentially huge amount of data (thousands of billions of bytes $\approx 10^{12}$) and linking a substantial amount of resources (memory in particular) to each processor. Only few tools (for both, modeling and implementation) are provided to close this gap in expectation, competence and education.

This article introduces the parXXL development environment, specially designed to close this gap between *fine-grained* modeling and *coarse-grained* computing architectures. It stems from two previous research projects that have investigated optimization of computing resources (CPU, memory, communications, synchronization . . .) and cellular oriented programming (to implement *fine-grained* models). Some collaborations with researchers in optic components and hot plasma (from LMOPS and LPMIA laboratories) guide parXXL design to an ease-to-use tool, and allow to identify collateral challenges. For example, some hot plasma simulation codes of our partners have been specially designed for global shared memory parallel computers. Intensive computation steps are split by optimized data rearrangement operations inside the global shared

⋆ In memoriam to our colleague Amelia De Vivo who passed away during the PARA-2006 conference on June 21st, 2006, in Umeå, Sweden.

memory. But on large distributed architectures this kind of operations would be prohibitive. Some codes need to be re-designed and based on local computations, in order to support efficient runs on large distributed memory architectures and straightforward *fine-grained* implementations with parXXL. We are convinced that this algorithmic and programming methodology is required to achieve large multi-scale simulations.

2 Related work

On the modeling side, Valiant's seminal paper on the BSP, see [1], has triggered a lot of work on different sides (modeling, algorithms, implementations and experiments) that showed very interesting results on narrowing the gap between, on one hand, fine grained data structures and algorithms and, on the other, coarse grained architectures. But when coming to real life, code developers are usually left alone with the *classical* interfaces, even when they implement with a BSP-like model in mind.

Development environment with explicit coarse grained parallelism, like MPI and OpenMP, usually lead to efficient executions but are not adapted to fine grained programming, and require experimented parallel developers. At the opposite, high level computation tools like Mathematica or Mathlab are comfortable tools to implement a simulation model from mathematical equations. But these high level tools have poor performances and use C++ code generators and classical distributed C++ libraries to create more efficient codes (an interface between parXXL and a Mathematica EDP solver based on cellular automata is under development). Some distributed OS managing a virtual shared memory like Kerrighed [2] allow to easily implement parallel algorithms, but they are limited to small clusters and require coarse-grained algorithms and optimized memory management to achieve performances.

Many generic cellular languages and distributed object libraries have been designed for parallel and distributed architectures, like Cape [3] or Carpet [4]. But they focus on *cellular automata* with cell connection limited to a predefined neighboring and with synchronous cell communications (similar to the *buffered* parXXL mode). Moreover, these researches seems to have slowed down since 2000. Finally, some Java based distributed environments exist, based on message passing and remote method invocations like ProActive [5] or on virtual shared memory like JavaSpaces [6]. Our personal experiments have exhibited good speedup and good scalability, and Java Virtual Machine performances are improving. But Java has not been yet adopted by the scientific community to implement intensive computations, and remains slower than C++.

So, implementing dynamic data structures (such as *cellular networks*) efficiently on a large scale often remains an insurmountable hurdle for real life applications. parXXL as proposed in this paper was created to lower that hurdle, in that it implements a general purpose development environment for fine grained computation on large parallel and distributed systems.

3 Software architecture

The parXXL development environment is split into several, well-identified layers which historically come from two different project sources, SSCRAP and ParCeL6. Its soft-

ware architecture is introduced on Fig. 1, and demonstrates the split of these two main parts into the different layers. The (former) SSCRAP part introduces all the necessary parts to allow for an efficient programming in coarse grained environments; interfaces for the C++ programming language, the POSIX system calls, tools for benchmarking, a memory abstraction layer and the runtime communication and control. The (former) ParCeL6 part introduces a *cellular* development environment and a set of predefined and optimized cell networks. These programming models of SSCRAP and ParCeL6 are detailed in the next sections.

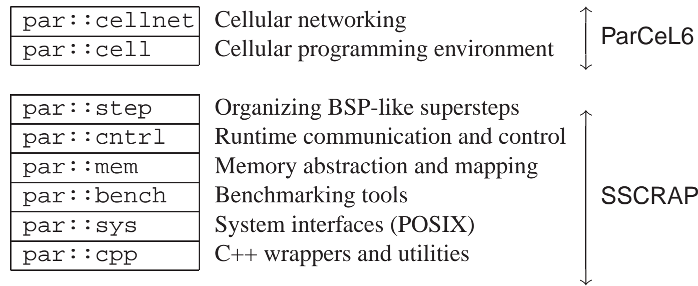


Fig. 1. parXXL software architecture

4 SSCRAP programming model

SSCRAP is a programming environment that is based on an extension of the BSP programming model [1], called PRO [7]. It proved to be quite efficient for a variety of algorithms and platforms, see [8]. Its main features what concern this paper are:

Supersteps with relaxed synchronization: Originally, BSP was designed with strong synchronization between the supersteps. PRO (and thus SSCRAP) allows a process to resume computation as soon as it receives all necessary data for the next superstep. The `par::cntrl` layer (see Fig. 1) implements these features in parXXL.

A well identified range of applicability: SSCRAP is clearly designed and optimized for *coarse grained* architectures. These are architectures for which each processor has access to a private memory that allows it to keep track of one communication to every other processor. If the platform has p processors, as a minimal condition this private memory must thus hold p machine words, a fact to which we refer as the architecture having "*substantially more memory than there are processors*". All modern high performance computing architectures (mainframes and clusters) easily fulfill this criterion.

Comfortable encapsulation of data: The work horse of SSCRAP is a data type (`chunk` in the `par::mem` layer) that encapsulates data situated on different supports such as memory and files which then can be mapped efficiently into the address space of the processes. Thereby SSCRAP can efficiently handle huge data (*e.g* larger than the address space) without imposing complex maintenance operations to the programmer.

Portability: SSCRAP is uniquely based on normalized system interfaces (`par::sys`), most important are POSIX file systems, POSIX threads and MPI for communication in distributed environments. Therefore it should run without modification on all systems that implement the corresponding POSIX system calls and/or provide a decent MPI implementation.

Performance: This portability is *not* obtained by trading for efficiency. In the contrary, we provide two run-times, one for shared memory architectures (threads) and one for distributed computing (MPI). These are designed to get the best out of their respective context: avoiding unnecessary copies on shared memory and latency problems when distributed. All this is achieved by only linking against the respective library, no recompilation is necessary.

5 ParCeL6 programming model

The `par::cell` level of parXXL architecture (see Fig. 1) implements the ParCeL6 *extended cellular* programming model [9]. It is based on *cells* that are distributed on different processors, and on a sequential *master* program: ParCeL6 developers design and implement some cell behavior functions, and a sequential program to install and to control a parallel cellular net. This mixed programming model is easy to use and facilitates the design of *cellular servers*: a classical client can connect to the sequential program, that runs cellular computations *on demand*. Main features of ParCeL6 cellular model are:

A dynamic cellular network: Starting from an empty network of cells, the sequential program creates cells on all available processors. Each cell has an individual set of parameters, and the first action of these cells is usually to connect each other to create a cellular network (a cell output can be connected to an unlimited number of cell inputs). This network is dynamic and may evolve at any point of the execution (cells and connections can be created or removed).

Six cell components: A cell is composed of (1) a unique *cell registration*, some (2) *parameters* and (3) *private variables*, (4) some *cell behavior functions*, (5) a unique multi valued *output channel*, and (6) several multi valued *input channels*. The first is imposed by ParCeL6 mechanisms, the others are defined by the developer.

A cyclic/BSP execution of the cell net: A ParCeL6 cycle consists of three steps: computation, net evolution and communication. Each cell is activated once during the computation step, where it sequentially reads its inputs, updates its output, and issues some cell net *evolution requests*. These requests define, kill, connect or disconnect some cells, and are executed during the net evolution step.

Three modes of cellular communications: During the communication step, *buffered outputs* are copied to their connected cell inputs. Their propagation is fast and is adapted to synchronous fine grained computation (cell inputs do not change during the computation steps). The propagation of a *direct output* to a connected cell input is triggered each time a *refresh* command is executed for it. This mechanism has a large overhead but is required by some asynchronous fine grained computations [9]. *Hybrid outputs* are an attempt to get both fast and asynchronous cellular computations: they propagate their value one time per computation step and per

processor (cells on different processors can read different values during one computation step).

Moreover, some *collector* mechanisms allow the cells to save data during their computation steps and the *master program* to gather, sort and store these data at the end of a computation step. Symmetrically, some *global* cell net communication mechanisms allow the sequential program to send input data to the cells (like camera images).

Finally, a classic ParCeL6 application code includes three main parts: the *sequential master routine* controlling the cell net installation and running cellular computation steps, the cell creation and connection operations to establish the cell net (can be easily implemented using the cell net library, see next section), and the cell computation functions that implement a fine grained computing model (like Maxwell equations, neural network computations. . .). So, a fine grained algorithm can be straightforwardly implemented on ParCeL6, especially when an adapted cell net template exists in the library, without dealing with parallel processing difficulties.

6 parXXL main functionalities

Optimized cell network library The `par::cellnet` library (see Fig. 1) is a collection of *cell network installers*: application code can easily deploy a cell network just using an *installer* object. Each *installer* has to be set with the application cell behavior functions, the cell parameter and cell variable types, and the cell network size. Then, it installs the cells and their parameters, and connects the cells according to a predefined communication scheme. Deployments are optimized: (1) the number of cells is balanced among the processors, (2) with preference neighboring cells are installed on the same processor, and (3) cell net installation is split into small steps to limit the memory required by the deployment operations.

The `par::cellnet` library currently includes network *installers* that place the cells on a 2- or 3-dimensional grid, and that will connect a given cell to all its neighbors that are ‘close’ with respect to a certain ‘norm’: e.g an installer of type

```
par::cellnet::mesh< 3, L2, 2, applicationType >
```

will position the cells on a 3-dimensional grid, and connect all cells that are at distance 2 in the Euclidean (L_2) norm. The remaining information of the particular application network (such as the individual cell functions etc) is specified via the type parameter `applicationType` that is defined by the application. Currently available norms are L_0, L_1 and L_2.

We are currently working on an extension of this setting to general dimensions. More generally, other types of regular networks (honeycomb, regular crystals) and other types of norms (e.g ellipses) may be implemented easily if needed.

Process specification Generally a parXXL program executes several parXXL-processes in an MIMD fashion. The kind of execution (POSIX threads or MPI processes) and the number of processes is not fixed in the code but only decided at link or launch time, respectively.

```
int define_cell(descriptor_t const& Descr, register_t& Reg);
int kill_cell(register_t const& Reg);

template< class TPARAM >
    int define_cell_param(TPARAM const& param,
                          register_t const& Reg);
```

Listing 1. `par::cell::context_t`, main cell management functions

According to ParCeL6 programming model (see Section 5), the `par::cell` layer restricts the MIMD execution model so that a `par::cell` program is composed of a sequential *master* program and a set of *worker* processes hosting and running cells. The *master* process installs and controls the cell net deployed on the *workers* using some high level cell management functions described above.

This *master-worker* architecture is adapted to many scientific computing applications, but *industrial* applications use *client-server* architectures. To support both scientific and industrial applications, the parXXL *master* process may also implement a *server* interface (installing a cell net, accepting client connections and running cell net computations *on demand*) and run on a specific *server* machine. The user can point out this machine at runtime among the pool used to distribute the parXXL program, using the `-s` option (`S`erver name). For example, on a cluster using the MPI parXXL runtime: ”`mpirun -np 100 MyAppli -s PE20 ...`” runs the application ”MyAppli” on 100 *workers* and installs the *master-server* process on the ”PE20” machine.

To avoid the *master-server* being overloaded by cell computations, the optimized cell net library `par::cellnet` allows to install or not to install cells on the *master-server* process, using the `-E` option (`E`xclude) at runtime. By default, the *master* process is the the parXXL-process 0 and hosts cells.

Cellular network management The main parXXL functions to easily manage a cellular network from the sequential program of the master process are shown in Listings 1 and 2. They are all members of the `par::cell::context_t` class.

Function `define_cell` allows to define a new cell, specifying its number of output values, its connection mode (*buffered*, or *hybrid*, see Section 5). parXXL defines its registration and its host processor. However it is possible to specify the host processor to optimize the cell net mapping; this is done by the `par::cellnet` library to create optimized cell networks. Usually the cells are defined by the sequential program of the master process, but they can be defined on any processor in parallel. This strategy will be exploited in the next version of the `par::cellnet` library to create larger networks faster. When some cells have been defined on one or several processors, function `conductCellUpgrade` executed on the master processor runs some processor communications and creates cells on different processors. Similarly, functions `define_cell_param` and `conductParamInstall` allow to define and associate some datastructures to cells (the cell parameters) and to send and install these parameters in the corresponding cell bodies on their host processors.

```
int conductCellUpgrade(void);
int conductParamInstall(void);

int conductComputation(ActivKind_t kind, size_t PermutIndex);
int conductLinkUpgrade(void);
int conductOutPropagate(void);
int conductCollect(size_t CollectorId);

int conductHalt(void);
```

Listing 2. `par::cell::context_t`, main cellular network management functions

The next group of functions is usually called in a *computation loop* exploiting the cellular network. Function `conductComputation` activates all cells on each processor for one compute cycle: each cell runs its current behavior function once. On a particular parXXL-processor the *order* of the cell activation may be specified: the order of their storage (default), its reverse, or in the specific order of a permutation table. In most cases, this order has no impact on the program result and has not to be considered in the design of the program. But some rare and *asynchronous* cellular programs are sensitive to the cell activation order (when using the *hybrid* communication mode). So, parXXL allows to quickly change this order to check the sensitivity of the program to this parameter using the parameters of function `conductComputation`.

Function `conductLinkUpgrade` allows to establish the cell links that were defined during the previous cell computation step. It generates some processor communications and datastructure update, that are mandatory to send cell output into connected cell input buffers on remote processors. Then function `conductOutPropagate` can route each *buffered* cell output to its connected input cell buffers, and the *hybrid* cell outputs can be routed automatically during the cell computation steps. Functions `conductCellUpgrade` and `conductLinkUpgrade` have only to be called when a the cellular network is established or changes during a cycle; in the common case that a cellular network is created and linked during the first cycles and fixed thereafter, they may be avoided once the network structure remains stable.

Function `conductCollect` is called to gather data that cells stored in a *collector* during the previous computation steps on the master process. Thus, *collectors* are distributed datastructures allowing to *collect* some results on the master process.

The last function, `conductHalt`, allows to halt all the processes excepted the master process running the sequential program. No parXXL function can be called after this function has been executed.

Memory allocation As already mentioned above, efficient handling of large data sets is crucial for a good performance of data intensive computations. The template class `par::mem::chunk` provides comfortable tools that achieve that goal. Its main characteristic is that it clearly separates the *allocation* of memory from the effective *access* to the data.

Allocation can be done on the heap (encapsulating `malloc`), at a fixed address (*e.g* for hardware buffers) or by mapping a file or POSIX memory segment into the address space (encapsulating `open`, `shm_open` and `mmap`). By default the decision between these different choices is left for the time of execution and can thus easily be adapted according the needs of a specific architecture.

Access to the data is obtained by an operation called *mapping*. Mapping associates an address for the data in the address space of the parXXL process and returns a pointer where the programmer may access it. When the memory is not used anymore, it will in general be unmapped. The template class `par::mem::apointer` provides a comfortable user interface that, as the name indicates, may basically be used as if it was an ordinary `C` pointer.

Mapping and unmapping can happen several times without the data being lost. In general it is much healthier for the application to free resources during the time they are not needed. In particular the system may chose to *relocate* the data at different addresses for different mapping periods, and will be thereby be able to react to increase or decrease of the size of individual `chunk` s.

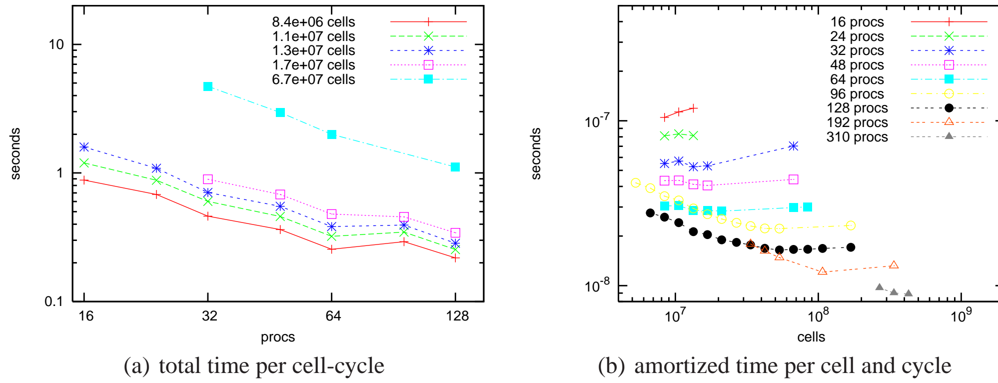
Mapping can also just request parts of the total memory (called *window* in parXXL). In such a way a program may *e.g* handle a large file quite efficiently: it may just map (and unmap) medium sized `chunks` one after another and handle them separately. Thereby a program may handle files that do not fit entirely into RAM or do not even fit in the address space of the architecture (4GiB for 32bit architectures).

An important property of `chunks` is that they may *grow* while they are not mapped. `par::mem::stack` uses this property for a simple stack data structure. This is intensively used by the `par::cell` layer to collect (and withhold) data during a computation phase on each processor before this data then can be communicated in its entirety in a communication phase. Thereby parXXL is able to avoid the fragmentation of cell communications into large numbers of small and unefficient messages (distributed architecture) or memory writes (shared memory).

7 Application and performance examples

Application introduction To start to validate the scalability of parXXL we have designed and experimented a 3D Jacobi relaxation on a *cube* of cells, with up to hundred millions of cells. This application has been implemented like a classical parXXL program. The sequential *master* process installs a cellular network on a pool of *workers*, and conducts a computational loop executed by each *worker*. There is no client-server mechanism implemented in this application, but it could easily be turned into a parallel Jacobi relaxation server with parXXL functionalities (see Section 6).

During the cell net installation steps, the cells are created with one output value, and are connected to their neighbor cells: up to six neighbors for a cell inside the cube. To easily deploy large cubes of cells, we have used the `par::cellnet` library, that installs optimized cellular networks (see Section 6). Then, the parXXL program enters a long loop of cell computations and cell output propagation (to the connected cells). The cells inside the cube update their output value with the average of their neighbor output values, while cells on the cube border maintain their output value unchanged.



(a) total time per cell-cycle

(b) amortized time per cell and cycle

Fig. 2. Benchmarks on Grid-eXplorer: up to 400 million cells using up to 310 processors

All the cell outputs of this application are routed to the connected cell inputs after each computation steps (*buffered* communication mode). At the end of the relaxation, some slices of the cell cube results are *collected* on the *master* process to be stored or displayed (see *collector* introduction in Section 5).

Experimental performances Fig. 2(a) shows for different problem sizes how execution time of a relaxation cycle decreases as a function of the number of processors. The benchmark platform is the *Grid-eXplorer*[1] cluster composed of bi-processor machines (a large PC cluster), running parXXL and its MPI-based runtime. These experiments exhibit regular decreases.

parXXL scales: On large enough problems, for the same problem size the time decreases linearly when increasing the number of processors.

However, the execution time of a complete relaxation cycle depends on the problem size (the number of cells): it has complexity $O(N)$, where N is the number of cells in the particular parXXL execution. To easily compare the execution times for increasing problem sizes, in Fig. 2(b) we show the average time to run a cell once: the execution time per cycle and per cell.

parXXL is robust: This time remains constant for a fixed number of processors, independently of the problem size or still decreases when running more cells on more processors (see the curves for 128, 192 and 310 processors).

By that, parXXL succeeded to scale up to 400 millions of cells and 310 processors on a PC cluster.

8 Conclusion and perspectives

Main parts of the parXXL architecture are implemented and operational, and first experiments show that the parXXL architecture scales up to some hundred processors for

[1] https://www.grid5000.fr/mediawiki/index.php/Orsay:Home

a large, fine grained application. Further development will consist in the following: (1) improve the *global communication mechanisms* to send data from the sequential program to the cells, such as camera images, (2) design and implement an efficient *hybrid cell communication mode*, (3) full generalization of the `par::cellnet` regular networks and a parallel deployment of these nets.

Future experiment will be run on a larger number of processors of the Grid-eXplorer machine, and on a Grid of clusters using Grid5000 (the French nation-wide experimental Grid). Our short-term goal by the end of this year (2006) is to deploy more than a billion of cells for simulations of laser-crystal interaction, a collaboration with researchers from the LMOPS laboratory.

To ease the implementation of this type of application from physics, a parXXL interface with a Mathematica PDE solver is under development (together with LMOPS). Our goal is to automatize the translation of a Mathematica code for PDE solving into a distributed large scale cellular computation. It will allow to speedup many research steps, avoiding long and tedious code translations.

Acknowledgments

Part of this research has been made possible via a visiting grant for Amelia De Vivo by the French programme ACI ARGE, and another part is supported by Region Lorraine. The experimental facet of this research is part of the Grid-eXplorer research initiative and relies on this experimental platform.

References

1. Valiant, L.: A bridging model for parallel computation. Communications of the ACM **33**(8) (1990) 103–111
2. Morin, C., Gallard, P., Lottiaux, R., Vallée, G.: Towards an efficient single system image cluster operating system. Future Generation Computer Systems (2004)
3. Norman, M., Henderson, J., Main, I., Wallace, D.: The use of the CAPE environment in the simulation of rock fracturing. Concurrency: Practice and Experience **3** (1991) 687–698
4. Talia, D.: 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 (2000) 595–603 Cancun, Mexico.
5. Baduel, L., et al.: Programming, Composing, Deploying for the Grid (chapter 9). In: Grid Computing: Software Environments and Tools. Springer (2006)
6. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces Principles, Patterns, and Practive. Pearson Education (1999)
7. Gebremedhin, A., Guérin Lassous, I., Gustedt, J., Telle, J.: PRO: a model for parallel resource-optimal computation. In: 16th Annual International Symposium on High Performance Computing Systems and Applications. (2002) 106–113
8. Essaïdi, M., Gustedt, J.: An experimental validation of the PRO model for parallel and distributed computation. In: 14th Euromicro Conference on Parallel, Distributed and Network based Processing. (2006) 449–456
9. Ménard, O., Vialle, S., Frezza-Buet, H.: Making cortically-inspired sensorimotor control realistic for robotics: Design of an extended parallel cellular programming models. In: International Conference on Advances in Intelligent Systems - Theory and Applications. (2004)