InterCell: a Software Suite for Rapid Prototyping and Parallel Execution of Fine Grained Applications

Jens Gustedt[1,3], Stephane Vialle[2,3], Hervé Frezza-Buet[2],
D'havh Boumba Sitou[4], Nicolas Fressengeas[4], and Jeremy Fix[2]

[1] INRIA Nancy – Grand Est, France
[2] SUPELEC – UMI GT-CNRS 2958, France
[3] AlGorille INRIA Project Team, France
[4] LMOPS laboratory, Metz University and SUPELEC, France

Abstract. InterCell is an open and operational software suite for implementation, code generation and interactive simulation of fine grained parallel computational models. This article describes the software architecture, some use cases from physics and cortical networks as well as first performance measurements.

Keywords: fine grained parallel models, interactivity, rapid prototyping

1 Introduction

The goal of the InterCell project is to help non-experts in parallel computing to use large scale parallel computers when developing models for physical phenomena, especially when these models have to be evaluated at large scale. To achieve this goal a software suite has been developed in order to allow rapid design and implementation of fine grained parallel computing models on coarse grained parallel architectures, *e.g* clusters or mainframes. The InterCell development cycle typically has several stages:

(1) rapid design of a mathematical model,
(2) automatic implementation of a fine grained parallel simulator,
(3) parallel execution of large scale interactive simulations, and
(4) large scale prototyping from the very beginning of model design.

Fine grained models of computation are widely adapted in different application domains. For our project this concerns two of these domains, namely the modeling of physical phenomena that have some notion of 'locality' (spatial or timely) and the modeling and development of neuromimetic networks [7]. But most likely InterCell could be useful for other domains as well.

Fig. 1 introduces the InterCell software architecture. At top level users describe their problems with application domain tools, such as a PDE solver or a cortical inspired neural network simulator. These high level tools generate fine grained parallel simulators, using a C++ library named Booz[5]. This library

[5] `http://ims.metz.supelec.fr/spip.php?rubrique27`

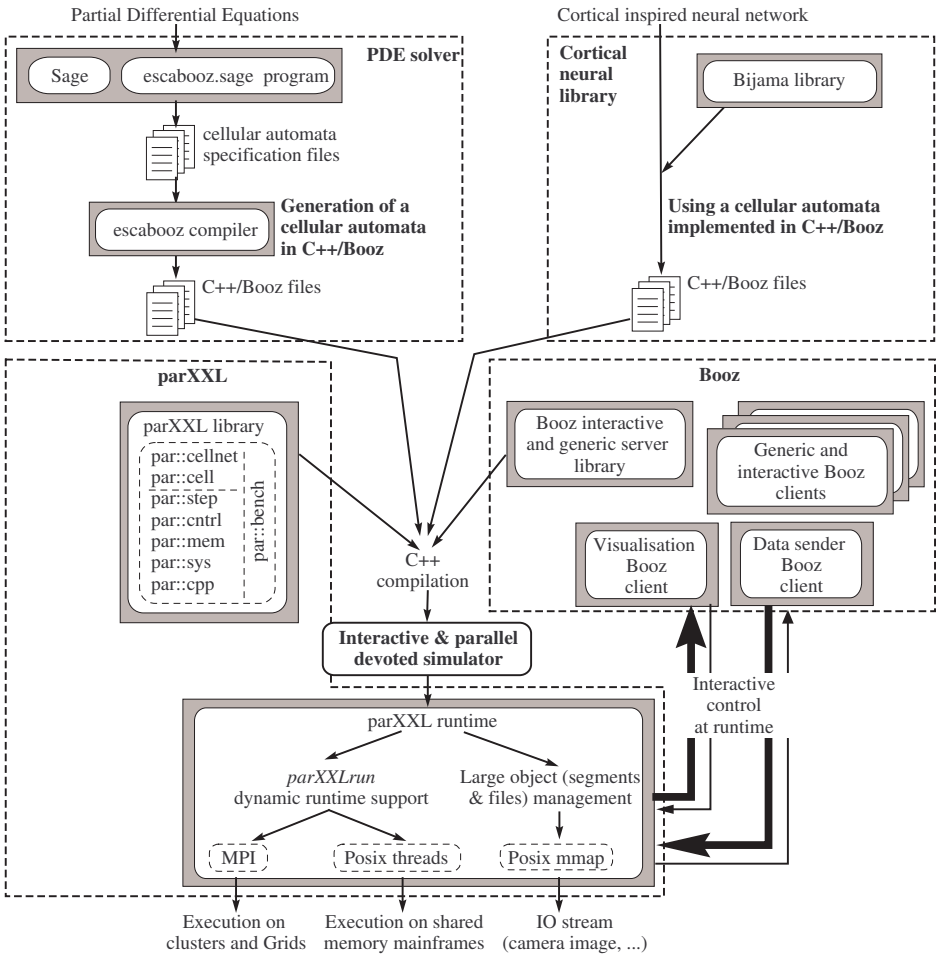


Fig. 1. Global architecture of our interactive problem modeler and PDE solver on large parallel and distributed computers

```
#Poisson's equation for semiconductor devices
eq1=(lmbda*nlap(phi(x,y),r,dr) ==
      ni*( exp(phi(x,y)) - exp(-phi(x,y))) - dop(x,y)).substitute(x=0,y=0)

#Newman contidions on one border
anp=(lmbda*nd2(phi(x,y),y,dy) ==
      ni*( exp(phi(x,y)) - exp(-phi(x,y))) - dop(x,y)).substitute(x=0,y=0)
```

Fig. 2. SAGE file (extract) specifying the electrostatic potential of a 2D P-N junction.

ensures the interactive control of the simulations and in turn uses the parXXL library[6]. With that, it efficiently maps the fine grained computations on coarse grained parallel architectures. The parXXL runtime hides the underlying parallel or distributed hardware. The final software has two parts: a parallel and interactive server that handles the actual computations, and a set of easy-to-use control and visualization clients.

2 Fine grained parallel computations

Fine grained computations that act on statically structured data (generally matrices) are nowadays well mastered and can be parallelized on coarse grained architectures (typically multicore clusters) with good results.

The case focused here is the computation on unstructured data for which the structure may even change occasionally and where the compute function that has to be executed may differ for each data point. Here an efficient mapping of computations to processors is not straightforward and good efficiency is generally difficult to achieve. parXXL provides a framework that facilitates programming under such constraints and draws good performances out of nowadays platforms.

The parXXL framework, see [5], includes several software layers, as shown in Fig. 1. Important for this project here are the following.

`par::cell:` a set of functionalities and a programming model to design and implement fine grained computations in the paradigm of so-called cellular computation. This layer allows to dynamically create and connect *cells* to establish cellular networks that are *executed cyclically*. When created, each cell is associated to four *cell behavior functions*: a function that is executed in each compute cycle, a query function that can be used to capture the state of the cell, a constructor and a destructor. A network of cells can easily be controlled by a sequential program, using *missions*, to create cells, execute one compute cycle, or extract data from the cells.

`par::mem:` an abstraction layer for handling large `chunks` of data. These allow for an efficient handling of large tables that are allocated on the heap or inside files and that can be resized dynamically. Technically, such a chunk may refer to memory on the heap (allocated with `malloc`), in shared segments (allocated through `shm_open`) or in files.
For the `par::cell` layer, it allows to group the cell data and output and access them in order or through hashed indices.

`par::cntrl:` handles the basic communication functionalities. It abstracts from the underlying runtime, currently MPI or POSIX threads. In particular important for this project has been the `transfer` family of functions that implements a `all_to_all_v` communication (used when each parXXL process needs to exchange different data with all other processes). In combination with the resizability of the `mem::chunk`, `transfer` dispenses to specify communication sizes and to allocate buffers beforehand.

[6] `http://parxxl.gforge.inria.fr/doxymentation/`

For the `par::cell` layer, these functions are mainly useful to implement cell communications, cell network creation and update.

`par::bench:` is used to instrument the library and to collect various performance data. In particular it registers the number of communications and their size, wall-clock and CPU times.

All these features are the foundations of parXXL, and have already been introduced in [5] and [4]. However, we have improved parXXL to support more asynchronous execution of cell networks, to easily collect and save results of large cell networks, and to increase its portability.

Cells can communicate the data from output to input channels in a synchronous or quasi-asynchronous mode. If in synchronous mode, cell input is updated at the end of each computation cycle. In quasi-asynchronous mode, cells are grouped in subsets and the output channels of the different groups are routed at different communication sub-cycles in the middle of the cell computations. So different cells reading a same output channel can read different values, depending on the concrete time of execution of their behavior functions in the computation cycle. The number of communication sub-cycles can be tuned at execution time. More sub-cycles lead to more asynchronism but to longer cell execution cycles.

In order to ease the extraction of results from large cell networks, we have designed some *collector* generic classes, and we have defined some new *missions* to easily run these data collection from a controlling sequential program. All these collections of large data are stored in chunks. These have been improved to map into files and avoid memory size limitations. Some optimized functionalities to read and write large data files storing N-dimension arrays of output channel values, have been added to parXXL. For example, they ease the initialization of large cell networks from data large files containing initial values for the output channels.

Finally, three parXXL runtimes exist: a first on top of MPI, a second on top of POSIX threads for multicore shared memory architectures, and a third on top of shared memory segments. All runtimes are available on 32 and 64 bits architectures. Moreover, great efforts have been made to improve the portability of our C++ template classes and functions, and our C++ meta-computing code. All these improvements make parXXL available and efficient on a larger set of parallel architectures.

3 Interactive parallel computations

One original property of the InterCell software suite is that cellular computation can be performed interactively. It allows visualization, writing and loading of snapshots, setting of cell values. This are all performed while the cellular automaton is briefly suspended. In addition it allows to step through the execution of the application. In our context that means that compute cycle after compute cycle may be observed individually. These features are provided by the Booz library that includes a visualization client, see Fig. 3.

This interactivity allows to use a cluster for situated systems, like robots, where cellular computation models the inclusion of an artificial brain in some real robot perceptivo-motor loop. It also provides a real-time view of the running process, that allows to detect convergence problems of the cellular models. Such an on-line availability allows programmers of cellular automata to prototype their model at a large scale, from the very first design stage. This is of primary importance since properties of large scale discrete dynamical systems are not easily predictable from small prototypes.

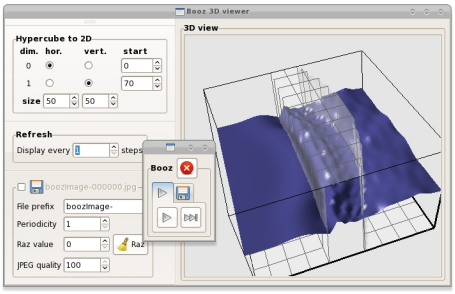


Fig. 3. Example of interactive Booz client

4 Examples of InterCell usage

Fig. 4 to 7 show examples of InterCell simulations. Fig. 4 is a classic 2D-Jacobi relaxation, where each *cell* simply computes the average value of its four neighbors. It has been implemented to test the functionality of our software suite. The application of Fig. 5 is modeled as a 2D grid of cells, each connected to its 8 neighbors. Here, each cell represents the elongation of a coil spring that is coupled to neighboring springs, in order to create 2D waves along the grid surface. The springs have different elasticity, the "lens" that is visible in light yellow shows the distribution of the elasticity among the springs.

The next sections, give the details of two simulations are real use cases of the InterCell suite, and that correspond to applicative research that was achieved in our laboratories.

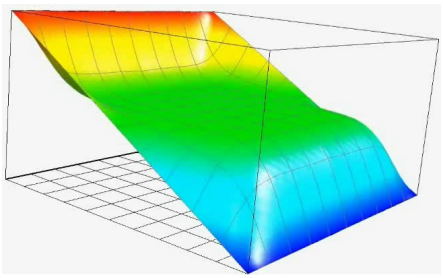


Fig. 4. InterCell simulation of a Jacobi relaxation

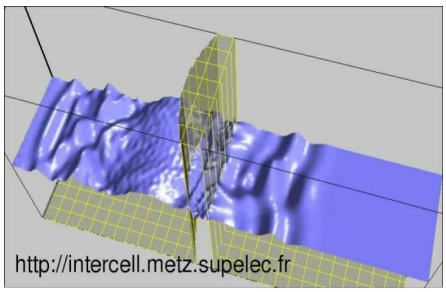


Fig. 5. InterCell simulation of a wave propagation

4.1 High level application codes

As illustrated in Fig. 1, applications are developed using high level programming environments and not the parXXL or Booz layers directly. The *semiconductor simulation* detailed in Section 4.2 has been implemented using only the SAGE programming environment. This allows to implement our PDE within a mathematical paradigm, easily. Our PDE solver module then automatically generates C++ source files that wrap Booz and parXXL functionalities. A final C++ compilation produces a parallel application running on any parXXL runtime.

The cortical networks detailed in Section 4.3 have been developed in C++, using our Bijama library (see Fig. 1). Again, the application developer focusses on the expression of his scientific models. For himself, he does not implement process creation, internode cluster communication, or process synchronization. However, the distribution of the neural network on the different cluster nodes is not yet fully automatic and requires some directives of the application developer. This issue in currently under investigation.

4.2 Modeling and Simulating a Semiconductor

Fig. 6 is a more complex simulation from semiconductor physics. It shows the result of a simulation of the electrostatic potential in a 2D P-N junction whose N-doped side is the square upper part while the P-doped part is the rest. This computation is done through the `sage/escabooz` part as described in Fig. 1.

The numerical method is based on a modified version of the Least Squares Finite Element Method (LSFEM), see [6]. From LSFEM, we have derived a "*local only*" recursive rule. It allows for each point in a mesh to be considered as an independent automaton. This is particularly well suited for fine grained parallel computing. The initial problem is a partial differential system of equations involving a Poisson equation and the field expressions from the doping of the material. Added to this system is a set of boundary conditions: Dirichlet type where ohmic contacts are present, and Neumann type elsewhere.

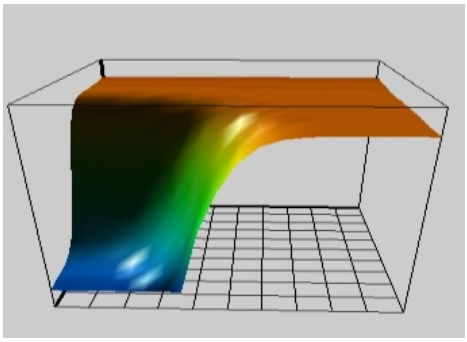


Fig. 6. InterCell simulation of a 2D P-N junction

The complete modeling and development process is thus as follows: the physicist (non-expert in parallelism) programs his equations in the SAGE[8] language, see Fig. 2, focusing entirely on physical and mathematical issues. Then, the `escabooz.sage` software suite, formally derives an update rule for each point of a given discretization mesh. Thereby it describes a complete cellular automaton. The SAGE program applies Newton's minimization method to a global error

term. This error results from a discretized form of the initial partial differential problem. Following Newton's method, an approximate solution is fed to the automaton. Once run in asynchronous mode, the automaton eventually stabilizes around a fixed point. This is the nearest minimum of the error term and corresponds to the solution to the discretized problem.

4.3 Modeling and Simulating Cortical Networks

Fig. 7 illustrates a second kind of application of InterCell simulations. In this simulation, we aim at studying the emergent properties of dynamic neural fields, a model of cortical neural tissue [1], in particular focusing on sensorimotor control of embedded physical agents (e.g. a robot). The interaction of perceptive, motor and motivational flows of information within the neural network allows the agent to interact on a physical world. The bio-inspired nature of this work requires to simulate large population of neurons that are permanently feeded by a perceptive input and that produce motor actions. To simulate the dynamic neural fields, the continuous equation (1) is discretized using the Euler scheme.

$$\tau \frac{du}{dt}(x,t) = -u(x,t) + h + \sum_{y} w(x,y) f(u(y,t)) + s(x,t) \qquad (1)$$

where $u(x,t)$ is the membrane potential of the neuron at position x and time t, h is a constant baseline, $w(x,y)$ is the kernel defining the interactions within the neural field and $s(x,t)$ is the input provided at position x and time t. The membrane potential $u(x,t)$ of all the cells evolves according to the same equation and therefore the computations are homogeneous across the cells. Simulating dynamic neural fields is, in this regard, particularly well suited for parallel implementations since a simulation usually involves large populations of fine-grained units. InterCell provides essential tools for scaling up models for realistic situations.

The simulation shown in Fig. 7 is a visual search task involving 11 2D dynamic neural fields, each made of 60×60 neurons (see [3] for details). The perceptive input is pre-processed along several dimensions (two colors, two orientations) and feeds a perceptive neural field. Specific connectivities within and between the fields lead to different emergent properties at the level of a neural field such as a competition between potential candidate targets, a working memory of targets that have been analysed or anticipatory mechanisms when camera movements are involved. On the bottom left of Fig. 7 is represented a visualization of the simulation with InterCell tools. The opportunity to visualize the whole network or a part of it is essential for tuning the parameters of the neural fields. In addition, InterCell allows to interact with the simulation on-line, constantly perturbating the network with a new perceptive input which is critical in the study of sensorimotor control.

The performance measurements provided in Section 5.1 were evaluated on a subpart of the model. This subpart involves three neural fields consisting of an input feeded by five stimuli, a competition neural field and a working memory.

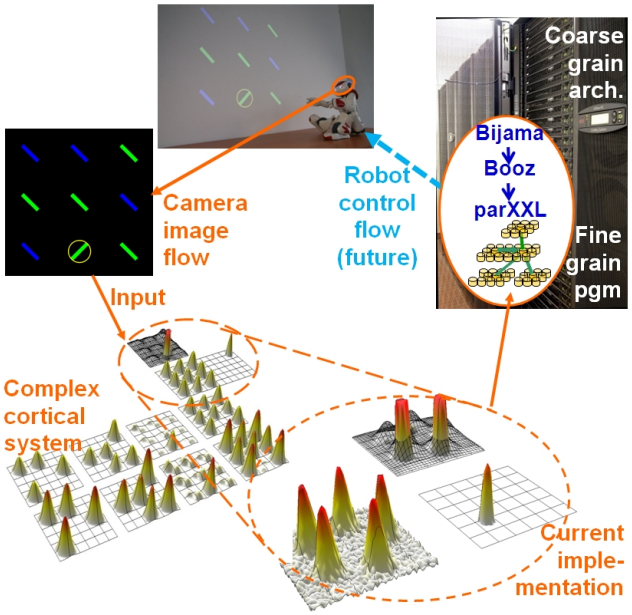


Fig. 7. InterCell implementation and run of a biologically-inspired neural network, for environment perception and robot control

The connectivity within this model is rather dense. It contains 10800 neurons with a total of around 30 million connections. InterCell easily handles large networks with dense connections while such a scale-up is hardly handled by a single computer.

The modeling and development process is as follows: the computational neuroscientist (not expert in parallelism) writes done the differential equations governing the evolution of the state of the neurons. This definition is written in `C++` with the `Bijama` library. It involves defining a step method for the units, the connectivity kernel, the parameters of the equation (e.g. h in eq. 1) as well as communication methods that allow to embed the model within the physical environment. Once these methods have been defined, InterCell automatically handles the parallel computations and communications and the situated agent can be controlled by a simulation running on a cluster without taking care of where and how the simulation actually runs.

5 Experimental performances

5.1 Performances of a wave propagation simulation

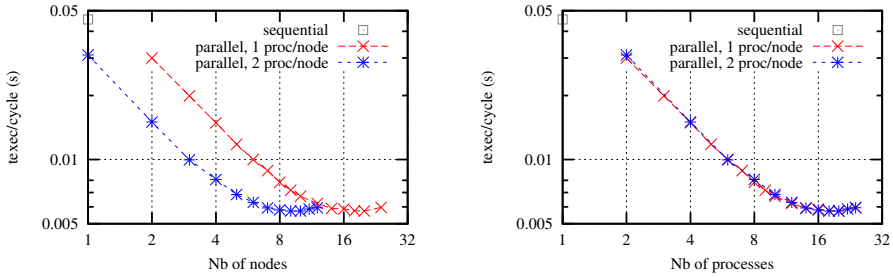
In Fig. 8 we show the results of a first performance evaluation of the second application of Section 4 on the InterCell cluster, see Fig. 5. This cluster consists of

256 2.66 GHz Xeon bi-core nodes that are connected via standard Gbit Ethernet.

The example application consists of a 100×300 grid of cells that is split evenly among the parXXL processes. We experimented several different splitting strategies to find out that (for this example) the difference in performance is negligible. Thus, here we only give the values for a split along the long side (300) of the grid. Thus the grid is divided into 1×2 parts, 1×3 parts etc, where each part is treated by a separate parXXL process. In one series of experiments we placed one parXXL process per compute *node* and in a second series we placed two, *i.e* one parXXL process per compute *core*.

Each data point in the figures represents an average over several runs of batches of 1000 compute cycles. We have chosen 1000 compute cycles per run, because it leads to execution times of our benchmarks from $5.7s$ to $45.5s$ in function of the number of parXXL processes used. These times are all several orders of magnitude greater than the precision of our time measurement tool. We did not observe significant variances of our measurements other than changing the number of process per core or node. So we concluded that the variation that was introduced by the OS or the interconnexion network was negligible. As a consequence, we only performed 3 runs per parameter set to do the averaging.

Plotted are the run times broken down to the time for one compute cycle for the network. Fig. 8(a) shows the time against the number of nodes, Fig. 8(b) the time against the number of parXXL processes. We see that both series show an optimal speedup in the range of 2–8 processes, and up to 16 processes the speedup is still reasonable. From thereon the addition of additional processes / nodes doesn't accelerate the computation. So for the given problem, a number of about 2000 cells per parXXL process is a reasonable minimal requirement. Fig. 8(a) also demonstrates that this setting is well suited to take advantage of the two cores in each node.



(a) As a function of the number of used computing nodes

(b) As a function of the number of run parXXL processes

Fig. 8. Execution time per compute cycle of the wave propagation simulation

5.2 Performances of a biologically-inspired neural network

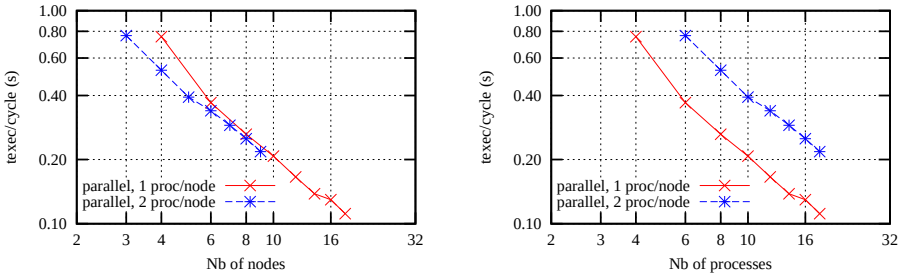
In Fig. 9 we show the results of a first performance evaluation of the biologically-inspired neural network application introduced in Section 4.3, see the bottom of Fig. 7. Again we used the dual-core nodes InterCell cluster to run our benchmarks, and a similar performance measurement approach.

This neural network system is composed of three cortical maps: a medium one and two large ones. Execution attempts on 1 or 2 nodes failed, because of lack of memory. We implemented the medium map on 2 parXXL processes running on 2 CPU cores located on 1 or 2 nodes (dual-core nodes), and we distributed each large map on 1 to 8 processes running on other cores and nodes. We realized two series of benchmarks with 1 and 2 parXXL processes per node. To fulfill the minimum memory requirements of the application, we had to launch at least 4 parXXL processes running on 4 nodes and 6 parXXL process running on 3 nodes, respectively.

We achieved a good scalability distributing the large cortical maps, and a speed up close to 6.8 using $18 = 2 + 2 \times 8$ nodes in place of $4 = 2 + 2 \times 1$ with one process per node, see Fig. 9(a). However, at the opposite of the wave propagation simulation, execution times are approximately 2 times longer when running two parXXL processes per dual-core node, see Fig. 9(b). More investigations are required to identify the contention: this might be due to a memory contention during computations steps, or a network contention during communication steps, or both.

6 Conclusion and perspectives

In this paper we presented InterCell, an open, operational software suite published under the GPL, see `http://ims.metz.supelec.fr`. This *development tool* is currently used by researchers in optics, photonics, and cortically-inspired neural



(a) As a function of the number of used computing nodes

(b) As a function of the number of run parXXL processes

Fig. 9. Execution time per compute cycle of the biological inspired neural network

networks. The later models are generally large and require interactive execution on large parallel systems. To these researchers, InterCell offers an easy-to-use tool to model and implement on a large, realistic scale. It provides automatic code generation and permits the parallel and interactive control of the simulation. First performance measurements are satisfying and show a good potential to address problems on a larger scale.

The next step in the development of InterCell will thus be to tackle applications of a larger scale: complex models are under investigations and large scale simulations are being implemented. Therefore, the Sage program that is currently used to specify the cellular automaton (which is still sequential) has to be parallelized to be able to process large problems rapidly. Also, some serial optimizations remain possible in the parXXL cell management.

Acknowledgments

The authors wish to thank Region Lorraine for its support to the InterCell project in the framework of the 2007-2013 CPER "Modélisations, informations et systèmes numériques (MIS)". Also we want to specially thank Patrick Mercier for his continuous support on the InterCell cluster.

References

1. Amari, S.: Dynamics of pattern formation in lateral-inhibition type neural fields. Biol Cybern 27(2), 77–87 (1977)
2. Boumba Sitou, D., Ould Saad Hamady, S., Fressengeas, N., Frezza-Buet, H., Vialle, S., Gustedt, J., Mercier, P.: Cellular based simulation of semiconductors thin films. In: Innovations in Thin Film Processing and Characterization - ITFPC 09. France Nancy (2009), http://hal.archives-ouvertes.fr/hal-00433062/en/
3. Fix, J., Rougier, N., Alexandre, F.: From physiological principles to computational models of the cortex. J Physiol Paris 101(1-3), 32–9 (2007)
4. Fressengeas, N., Frezza-Buet, H., Gustedt, J., Vialle, S.: An interactive problem modeller and pde solver, distributed on large scale architectures. In: Third International Workshop on Distributed Frameworks for Multimedia Applications - DFMA '07. IEEE, France Paris (2007), http://hal.inria.fr/inria-00139660/en/
5. Gustedt, J., Vialle, S., De Vivo, A.: The parxxl environment: Scalable fine grained development for large coarse grained platforms. In: PARA-06: Worshop on state-of-the-art in scientific and parallel computing. vol. 4699, pp. 1094–1104. Sude Umea (2007-09), http://hal-supelec.archives-ouvertes.fr/hal-00280094/en/
6. Jiang, B.N.: The Least-squares Finite Element Method: Theory and Applications in Computational Fluid Dynamics and Electromagnetics. Springer (1998)
7. Ménard, O., Frezza-Buet, H.: Model of multi-modal cortical processing: Coherent learning in self-organizing modules. Neural Networks 18(5-6), 646–655 (2005)
8. Stein, W.A., et al.: Sage Mathematics Software (Version 4.3). The Sage Development Team (2009), http://www.sagemath.org