Rekindling Parallelism

Frédéric Gruau*†
* Laboratoire de Recherche en Informatique
Université de Paris-Sud 11, France
Email: gruau@lri.fr

Fabien Michel†
†Laboratoire d'Informatique de robotique et de Microélectronique
de Montpellier - Université Montpellier II - CNRS, France
Email: fmichel@lirmm.fr

***Abstract*—Computing in parallel means performing computation simultaneously, this generates two distinct views:**

- ***Performance view* A mean to accelerate computation using coarse grain parallelism.**
- ***Decentralization view* A new way of programming by decentralizing massive fine grain parallelism.**

Researchers on massive parallel models study the programming *expressiveness*, i.e. new bio-inspired ways of computing such as artificial neural network or multi agent systems solving new kinds of problems, but are usually not directly concerned about high performance. In contrast, researchers on high performance tend to narrow the scope of parallel expressiveness by preserving the sequential model of computation and defining specific language constructs that can lead to parallel run-time *performance* for more classical parallel algorithms. We argue that parallelism will really fully blossom only when both views get unified through the achievement of a new generic computing model that, while enabling decentralized computation, also supports classical way of programming and incorporates the hardware constraints to provide parallel performance. We are working on such a generic model called *self developing self mapping network*. This paper first justifies the motivation for such a model, and then sketches the fundamental principles of this model.

***Index Terms*—parallel computing; cellular automata; decentralized computation**

I. Overview

In section II, we analyze the overwhelming presence of the sequential way of programming parallel computer. In section III, we formulate what is the additional expressiveness that justifies and promotes decentralization as the real essence of parallelism. A formal property of state reachability is put forward. We show the advantages of decentralization using three examples: Multi-Agent Systems (MAS), the brain, and Artificial Neural Networks (ANN).

The rest of the paper discusses the challenges that need to be met so that decentralized parallelism also integrates a more classic structured programming framework, with the high performance recompense. Section IV first shows that the difficulty of programming massive parallel systems could be related to the fact that dynamic structures cannot be instantiated, and thus proposes the *Self development of network* as a computing model designed with this issue in mind. Section V considers performance issues, analyzes how high performance is obtained within the sequential dogma, and highlights the underlying scalability problems. We make precise the mapping problem that needs to be solved when massive parallel hardware is targeted and introduce a new solution called self-mapping, which can be applied to self-developing networks.

II. High Performance Parallelism as a Continuation of the 'Sequential Dogma'

Our programming abstraction, as well as the hardware of our computers reflects a sequential dogma:

- *Software* We think a step-by-step sequence of instructions that modifies a global state,
- *Hardware* We partition the hardware between a very big passive part: The memory that stores this global state and a hyper active processing part: The processor, which executes that instruction flow.

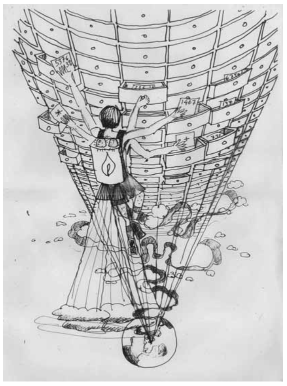

Fig. 1. Artistic view of the Von Neumann's avatar, courtesy of P. Femenias.

The sequential dogma is well accepted, but may look quite bizarre, as rendered in Fig. 1. This dogma was adapted to the early days of computers because of the scarce hardware resources available at that time: The first processor had only 2250 transistors. What now that this number has reached several billions? To exploit these huge resources, modern processor chips include devices enabling to compute many instructions simultaneously. Parallel processing has replaced sequential processing. A chip is now hierarchically organized, divided in several computing cores, each core having many functional units, each functional unit processing hundreds of bits simultaneously. Memory is also a hierarchy from register to cache and hard disk. However, the sequential dogma, as we stated it, is still almighty. Everyday programming is still about a step-wise modification of a coherent global state. The hardware is still divided between memory and processing. Parallelism is commonly understood as an additional shortest-as-possible set of constraints put on the programming style that, if fulfilled, will enable a parallel compiler to produce code that exploit the hierarchical architecture and lead to best-as-possible performance. In short:

Definition 1: Computing in parallel means computing in sequential, but troublesome and faster.

The sequential dogma's robustness can be explained by its low cost, and its suitability to current social needs. Giving up the global state hypothesis means programming a decentralized system made of many interacting parts. Such complex systems are much more difficult to program, or even to harness. Furthermore, our usual applications are not naturally decentralized. The sequential dogma is a key hole through which we look at parallelism because it considers only a tiny bit of the spectrum of computation that can be run on hardware, ignoring the natural expressiveness of a parallel computing medium. Let us now try to characterize this expressiveness.

III. Natural Expressiveness of a Compute Medium

A. An Alternative to the Sequential Dogma

As a set of interacting computing devices, a parallel machine is usually controlled and programmed top-down to compute faster a given well defined high level task. This misses an important point: If we reason bottom-up, and directly study such a set of interacting devices as a computing model, each device can maintain a local state, and be programmed in a decentralized way allowing it to compute differently rather than simply "acceleratingly". Performance can also be tremendous, because it can scale with the number of devices, thanks to decentralization. This shift of mindset on parallelism arises by considering it as the aggregation of basic resources available in massive quantity: An arbitrary large number of smallest as possible devices. The smallest computing device is a Finite State Automata (FSA). The resulting theoretical model of a parallel machine is thus a network of automata that can each read the state of adjacent automata and update their own local state. Arbitrary scalability imposes some further locality constraint on the interconnection between automata: We consider a natural way to obtain it, which consists in distributing the automata homogeneously in euclidean space, and establish connections locally in space. Three common examples are Field Programmable Gate Arrays (FPGA), Cellular Automata (CA), and sensor networks. Those architectures make the object of study of the *spatial computing* community [1] that refers to them as *computing media*. In a computing medium, there is no dichotomy between a huge passive memory part and a hyperactive small processing part. When programmed in a decentralized way, each automaton potentially takes any of its local states, partially independently from its neighbors. If each automaton can choose between 2 states, a network of n automatons can choose in one step, between 2^n states. This actually lead to a formal definition of decentralization.

Definition 2: A computer architecture is decentralized if the number of states that can be reached in one time unit grows exponentially according to its size.

The potential number of states stored in a memory grows exponentially with its size, but classically, a memory only stores a *passive state*, e.g. a collection of data. In a sequential program, a single memory cell id changed at each time step. The number of states accessible from a given state is linear in the memory size, instead of exponential. In practice, sequential programs tend to cycle through a small number of well designed states, which remain proportional to the program's length. This is illustrated by the concept of flowchart which summarizes all the possible states. In contrast, because each automaton updates itself in parallel, a point in the state space of an automata network is an *active state* like an enormous program pointer. In an ANN, this program pointer follows a rich dynamic which is the computation itself, whereas in sequential programs, the program pointer stubbornly cycles through the same bunch of loop nests and recursive calls.

Forcing an automata network to go through a specific sequence of global states, is a misuse considering its great expressiveness: Automata would have to behave in a predetermined centrally organized way and would not be independent any more, collapsing the state space. So, if we are interested in preserving the intrinsic state richness property, we obtain the alternative definition for parallelism:

Definition 3: Programming in parallel means programming a decentralized system with a style enforcing an exponential number of reachable states in one system transition.

If each automaton had a single choice. the number of reachable state would be one, therefore, a decentralized system needs automata to be non deterministic. On the other hand, if an automaton behaved totally independently from its neighbor, it would prevent a coherent collective computation to go on, therefore the independence of an automaton with respect to its neighbor should be only partial.

B. Examples of Decentralized Parallelism.

Engineering Application Specific Circuit, or FPGA circuits for digital signal processing is a common industrial application of automata network. The pipelined parallelism implicit in this domain can be exploited by laying out circuits of operators processing a flow of identical units of computations. However this programming approach is not decentralized. We distinguish two types of successful decentralized programming MAS and ANN. In MAS, automata are embodied in agents that can move in the medium, reproduce or die, and interact with many other changing neighbors, so the virtual network allowing automata to exchange messages is not fixed. In contrast, ANN neurons are wired into circuit, thus communicating with the same neighbors for a long duration.

1) Multi-Agent Systems: MAS is a modeling [2] and programming paradigm [3] which underlying philosophy strongly relies on decentralized parallelism. Each agent is modeled as a computational unit which perceives and acts locally in an environment, according to its own internal clock. Although MAS can therefore benefit from parallel hardware, they have always been priorly considered for the computing features arising from their decentralized nature: Quoting the pioneer work on *the Distributed Vehicle Monitoring Testbed* [4] :

> Distributed problem solving also differs from much of the work in AI because of its emphasis on repre-

senting problem solving in terms of asynchronous, loosely-coupled process networks that operate in parallel with limited interprocess communication.

Decentralization is today the foundation of a whole trend of research relying on a bio-inspired approach similar to ANN's, but from a collective perspective. Related researches are inspired by the structural and functional aspects of natural MAS (e.g. ant colonies). As thoroughly explained by Parunak in [5], such MAS not only fulfill global functions but also exhibit robustness and adaptiveness thanks to their very nature:

Firstly, at the micro-level, each agent is *small in (1) mass, (2) time, and (3) scope*: (1) An agent is negligible compared to the whole system so that its performance variations do not destabilize the system: Collective dynamics prevail. (2) *Agents are forgetful*. In such systems, agents produce, exchange and deal with information which is made ephemeral by environmental dynamics: As pheromones evaporate, obsolete ant paths disappear rather than misleading other colony members, thus making the whole system adaptive. (3) *Local sensing and actions*: Agents cannot and do not need to know the global state of the system: *Control is done from the bottom up*. Agents rather sense and act only on their immediate vicinity so that they compute and behave only according to a small set of data.

Secondly, at the macro-level, the behavior space of the entire system is huge compared to the programming effort so that massive MAS fall under the definition 3 given earlier. MAS are able to generate numerous different trajectories, especially because of non-determinism in agent decisions and nonlinearity of interactions [6]. Roughly, for 100 agents with 10 behaviors, if the system's state transition function is computed by having each agent acting once, the number of reachable system states is on the order of 10^{100} for each transition [6]. Furthermore, such systems are fully decentralized: There is no central agent. If such agent would exist, it would be both a single point of failure and a potential performance bottleneck for the system. As highlighted by Parunak, many transversal research works emphasize that this feature is the fundamental key that enable diverse complex systems to efficiently fulfill their purpose.

So, from a high level perspective, the computation which is globally done by natural MAS is performed horizontally: It is the interactions among the agents, and how they are structured thanks to environmental dynamics, that produce the global behavior. Moreover, these systems efficiently fulfill global functions, while showing robustness and adaptiveness to dynamically changing circumstances, precisely because they are decentralized [5], [7], [8].

2) The Brain: A good large scale example of existing decentralized automata network is given by nature: The brain contains roughly $15 - 33$ billion neurons, linked with up to 10,000 synaptic connections each. Execution involves a stirring of every bit of information which is continuous, decentralized, non-deterministic and massively parallel. The memory part and the computing part of each processing element are intimately mixed to the point that they cannot be separated. The brain has the ability to perform a wide variety of tasks (programmability), involving sensing and actuating in the real world, which are very difficult for a computer. Brain's computing spectrum can be termed *general-purpose in real world*. Here is an example of difficult task solved with a high degree of parallelism in a decentralized way: Experiments show that the brain can decide if an image contains an animal in less than 150 ms [9]. Since it takes approximately 1 ms for a neuron to fire a spike, electric signals starting on the retina bounce back and forth between neurons about 150 times. Given the complexity of the task and the elementary nature of neuron processing, the computation undoubtedly implies a huge number of neurons simultaneously, i.e. a significant portion of the gross 10^{15} synapses. In contrast, the parallelism exposed in sequential programming style is usually no more than the size of the array data structure used in the program, but the processing involves millions, if not billions of stages. Assume we draw computation and data (or signal) movements so that simultaneous computations are drawn on the same horizontal line and data dependencies are edges going up vertically: Each node takes 1cm^2 of paper, where the height unit corresponds to a gate or neuron switching time constant. A one second run of a high performance computer would then fit on a sheet of paper, a few meter width, but with an height easily reaching the moon. In contrast, a one second run of the brain would be only 10 meters high, while wrapping around the earth several times. High performance computers compute vertically while the brain computes horizontally. This indicates the existence of a decentralized way of computing that is different in means and goals.

3) Artificial Neural Networks: ANN form a computational model inspired by the structure and functional aspects of the brain. It is a specific instance of automata network. Each automaton models a simplified biological neuron. For example, it can make a weighted sum of inputs, and applies a threshold function. ANN process information using a connectionist approach to computation, which stresses the parallel nature of neural processing and the distributed nature of neural representations. It is worth noting that connectionism was originally known as *Parallel Distributed Processing (PDP)*, which was a popular terminology in the 1980s with the milestone book [10] by McClelland, Rumelhart and the PDP Research Group. This shows that at that time parallelism was indeed considered as a different rather than faster way of processing.

C. Using Analog Time: A Core Step toward Decentralization

We advocate that there is one stepping stone toward decentralization: Use the hardware devices in their analog regime and *Abandon the clock!* The resulting model is yet more difficult to program with, but has yet considerable more intrinsic power. Relaxing the clock enables each device to independently choose the precise moment of its activity. Time is analog, and comes free at an infinite precision, which is effectively used by the brain. Neurons' firing are interleaved in time, and the precise moment of firing plays a critical role. This is believed to be a key factor accounting for visual discrimination speed [9]. Developers see time scheduling as a heavy constraint requiring time-expensive mechanisms such as barrier synchronization, nature deals with it as a basic ingredient useful to compute with and speed up computation.

IV. Programming Computing Media

A. What Makes a Computing Medium Hard to Program

Fascinating they may look as brute computational force, raw computing media such as a FPGA, or CA, have never been popular as a computing model. They are inherently hard to program for general purpose computing, no compiler from some high level computing language exists: ANN are not compiled from algorithms. FPGA are compiled, but using a language describing circuits instead of algorithms. We argue that the key impediment is the static topology: A given point in the medium, an automaton, always takes its inputs from the same neighborhood. The automata network is fixed. In the connectionist approach to ANN, each neuron remains connected to the same neurons. In FPGA, parts of a given circuit are usually not dynamically reconfigured even if it is within technological reach. A fixed circuit cannot express parallel algorithms with dynamic data dependence such as Quicksort. Programming a fixed circuit to compute an algorithm needs a very specific mindset and is mainly adapted for signal processing. The brain itself is not a static circuitry: It evolves during embryogenesis, after birth, and in fact, throughout the entire life time.

The power of a programming language is to control the dynamic unfolding from structures, be it with the stack or the heap, be it for the data structure, or the hierarchical function calls. We argue that the basic framework of automata network misses a similar dynamic property that would free the programmer from the burden to specify individually what each particular point of the medium should do. Our goal is to relax the static constraint, in order to enable the programming of parallel complex systems. We propose to consider a network structure that can dynamically develop. Of course, it will not be the hardware that grows new connections and new processing nodes, but a software system layer placed right on top of the medium, making it look like developing.

B. Self Developing Machine

Self development machines are automata networks *with a programmable topology*. New automata and new connections can be added at run time. Each automaton must do more than compute its next state: It should also specify an action that adds a new node or deletes itself, or modifies its local connections as well. This is formally known and studied as *graph rewriting*. The repertoire of actions is fixed. The automaton chooses its output action together with its next state. Formally, this is a *mealy machine*, and since the execution of the actions produces a development, we call it a *self developing machine* (SDM). For clarity, it is convenient to distinguish the network nodes from the automaton. A node is called *self developing agent* or more simply *agent*.

A canonical initial configuration consists of a single ancestor agent connected to a set of fixed agents doing the parallel input and output to an external host. The ancestor generates new agents and connections, and develops a network. Each agent runs the same automaton, but has its own state. Unlike automata network or computing medium, self development is a computing model, not a hardware model. The primitives are designed so that executing a SDM on a computing medium is feasible. A self developing agent is simulated on the medium by a connected set of processing elements called its support. Two adjacent agents must have adjacent supports. The creation of new agents is done by dividing the support of an agent. The different rules implementing preservation of support, or division, and communication can be implemented as a system layer on a computing medium, transforming it into an easier to program virtual SDM. Besides, the agent's neighborhood integrates the hardware constraint, e.g. for the reasonable case of a 2D computing medium, the network must remain planar.

C. Self Development as Programming Model

The fine-grain nature of each agent -lighter than a thread- is an important aspect distinguishing self development from multi-threading. Lightweight is required so that moving constantly agents across the computing medium is feasible. Conceptually, each agent should carry an elementary piece of data: A single scalar value, and an elementary piece of code telling how to use its data. So an agent cannot compute anything by itself and needs to communicate and combine its value with the ones of its neighbors. A non-trivial computation can take place only by unfolding a network and communicating data along the network edges. So programming a SDM really means programming a dynamic network.

As a beginning, one can work out a small finite state automaton (FSA) to develop generic architectures reflecting computer science data structures, such as binary trees or 2D grids. Computation takes place when agents communicate along the dynamic network, which can happen during development or after. We showed in [11] that complete parallel algorithms such as sorting or matrix algebra, can be compactly compiled into a single FSA from a higher level language description. In contrast to the sequential dogma, the resulting programs are an homogeneous description that does not separate the code from the data stored in an external memory.

V. High Performance on Computing Media

A. Stating the Problem of Massive Parallel Performance

1) The Sequential Dogma Promotes Architectures that are not Arbitrarily Scalable: Preserving the sequential dogma is the major force shaping the silicon of industrial chip market, at the expense of scalability. As a result, shared memory machines are the underlying model of today multi-core processors, because it fits the sequential dogma by preserving the model of a Uniform Memory Architecture (UMA). We are interested in long term massive parallel hardware, and shared memory is well known to not scale very well above tens of processors [12]. The alternative is to use distributed memory machines. Graphics Processing Units (GPU) card is a promising example offering an impressive computational power. However, distributed memories are in general always implemented together with an all-to-all router, which goal is to allow any two PEs to communicate, using processor's id. Two arbitrary PEs are considered as being close together. But the performance of these features cannot be scaled either, because the time required for a signal to travel the length of the wire is not taken into account.

2) Defining Computing Media as Scalable Architectures: We therefore consider only distributed memory machines with no global router: Each processor communicates only with its direct neighbors. Thus the network topology must be taken into account. Communication is done in constant time between neighboring processors. Communication between remote processors should be relayed through a path of intermediate processors. Pushing scalability to millions of processors put more constraint on the architecture: One must evaluate performance according to the *VLSI model of complexity* stating that it takes one unit of time for a signal to travel one unit of distance. A typical scalable architecture on a 2D chip is the 2D grid. [13] shows that any regular grid in 2D space complying with VLSI complexity is similar to the 2D grid in some sense. One can go further and considers scalability above the million of processors, by relaxing the clocked behavior, and the crystal regularity of the arrangement. Amorphous computers [14] are made of loosely couple computing devices, homogeneously scattered in 2D or 3D space. They should be of sufficient density, so that local radio signals sent by a device reach an average of 15 surrounding devices. Amorphous computers are mainly used as distributed sensor networks. But they can also simulate discretized physical laws like CA [15], making them a plausible target architecture for self development, since the parallelism is precisely based on simulating simple physics. The term *computing medium* used throughout this work refers precisely to those scalable architecture, regular 2D grid or amorphous computers, that discretize space.

3) Mapping a Parallel Model to a Scalable Parallel Hardware: Consider a scalable architecture, which is as discussed, a network of distributed memory processors. The performance of the mapping depends on two conditions:

- *Load balancing:* Balance the number of computations done by each processor.
- *Communications:* Match the dependence graph with the processor network.

Load balancing is necessary to maximize processor usage. E.g. when synchronization barriers are used, and one PE gets double of work than others, everybody have to wait for this one. Communication is often the bottleneck for performance. To minimize communication, computations depending on each other should preferably map on the same processor, or at least on neighboring processors. The idea is that if the dependence graph has to be sliced into clusters of dense connectivity, each cluster's nodes should be mapped to the same processor, and adjacent clusters should map on adjacent processors.

B. Existing Mappings toward Scalable Parallel Hardware

We now review the different ways in which the mapping from software to hardware tackles the main two problems mentioned: load balancing, and communication minimization.

1) The Mapping Underlying the Data Parallel Approach: Data parallelism proposes to solve the mapping by distributing data and applying variations of the *owner compute rule*, stating that computations using a data are done by the processor holding this data and computations are proportional to data.

- *Load balancing:* Arrays are tiled into regular blocks, and distributed homogeneously.
- *Communications:* The lattice of dependence is projected to the lattice of processors.

A loop nest is called regular when all the array accesses are affine expressions of the loop indexes. The dependence graph has a crystalline many dimensional lattice structure that can be automatically projected on a regular lattice of processors such as the 2D grid. This approach can provide massive parallel performances that scale. However, if the program holds several loop nests, the underlying optimization problem of remapping arrays across loop nests is NP complete in the number of loop nests [16]. Furthermore the approach limits the parallel expressiveness, e.g. functional or heterogeneous pipelined parallelism cannot be addressed.

2) The Mapping of Generic Automata Network: Automata networks such as ANN provide an interesting intermediate stage of mapping compared to generic dependence graph, and can be seen as a folded form of a dependence graph where each automata performs many computations of the underlying dependence graph. Assuming that the overall computation contained in n node of a given dependence graph, in evenly distributed between m automaton, where $m \ll n$ each doing n/m computation, the size of the mapping problem is effectively reduced from n to m. The mapping is now formulated using this coarser grain building block:

- *Load balancing:* Balance the number of automata mapped to each processor.
- *Communications:* Match the automata network with the processor network;

This matching means that automata which are neighbors and need to communicate must be mapped to either the same processor, or at least to neighboring processors. It is difficult to check these two conditions, the structure of the automata network does not necessarily correspond to the processor network structure. Even in the case where it would match, the actual finding of the match between two graphs, called graph homomorphism, is a NP hard problem, not solvable on large scale. This difficulty explains why ANN are not famous for efficient distributed implementation. In the literature, we found mainly examples where only the ANN's global structure is taken into account to partition it into clusters [17].

3) The Mapping of Pipelined Circuits on FPGA: Reconfigurable platforms perfectly fit our definition of scalable parallel hardware. Parallelism is usually obtained using pipelined circuits made of operators that process computation streams. The mapping problem is called placement and routing: The operator with lowest throughput is bottlenecking the global throughput. Therefore, more hardware should be devoted to it to improve the overall performance:

- *Load balancing:* Speed up the least bandwidth operator.
- *Communications:* Minimize the longest wire.

A VLSI circuit is hierarchically organized using library elements, reducing the problem size and making the mapping more feasible. One circuit mapping technique of interest is called force-directed placement [18] and consists in optimizing a random initial placement by simulating physical laws: Connections between nodes act as springs, pulling nearer any pair of communicating automata to reduce communication latency.

C. The Self Mapping of Self Developing Network

The mapping of self developing network is done using a generalization of force directed placement:

- *Load balancing:* All the agents exert a repulsive force
- *Communications:* Connected agents exert an attractive force

Repulsive forces between neighbor agents in space lead to an homogeneous distribution. Until now we have implemented only repulsive force and homogenization. The 1D case is simple enough (9 bits of state) to prove the convergence [19]. The 2D case needed around 200 bits states and has not been published yet, but a Java applet shows the homogenization rule in action for different initial situations [20]. Because agents are gradually added, there is no need to deal with a tangled random initial situation. The initial ancestor agent starts in the middle of the computing medium, and the input/output agents are fixed on the border. An agent is allowed to compute only when the forces acting on it are locally in equilibrium. Whenever new agents are added, the computing medium takes the control, simulates the forces, and moves the agents accordingly until a new equilibrium is reached and then gives back the control to the agents. Since the network is a planar graph, the adjustment needed at each step is sufficiently simple so that each agent is directly attracted toward its new equilibrium position. This secures speed, and prevents the apparition of local minima.The parallelization effort is shared between the user specifying the self development and the machine doing the dynamic mapping. There is no need for a treatment in-between, such as a complex mathematical analysis, an intelligent compiler, or a placement and routing software. The simulation of the force is done throughout the same computing medium. The elementary computing resources holding an agent can also simulate the forces throughout the space using a CA discretization of physical laws. Mapping is thus parallelized in an homogeneous way with the computation itself. The mapping is considered as a central run-time hard task rather than an offline static optimization. Consequently, it is allocated a fixed percentage of the processing power of each PE.

VI. Conclusion

This paper is a discussion about parallelism: It proposes an analysis of the actual situation and advocates for research directions that consider decentralization as a core feature of parallel hardware. Such a perspective does not necessarily means abandoning the traditional modular programming approach. However, going on with traditional programming with such a different context requires new principles enabling structuring a raw computing medium into an easier to program virtual machine. Considering this purpose, we sketch a model called *self developing self mapping network*. This model now needs to be turned into a working simulator which will be used both as a proof of concept and as an empirical study tool.

References

[1] A. Dehon, J.-L. Giavitto, and F. Gruau, Eds., *Computing Media and Languages for Space-Oriented Computation 2006*, Dagstuhl international workshop 06361, 2006.

[2] F. Michel, J. Ferber, and A. Drogoul, "Multi-Agent Systems and Simulation: a Survey From the Agents Community's Perspective," in *Multi-Agent Systems: Simulation and Applications*, ser. Computational Analysis, Synthesis, and Design of Dynamic Systems, D. Weyns and A. Uhrmacher, Eds. CRC Press - Taylor & Francis, 05 2009, pp. 3–52.

[3] J. Ferber, *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] V. R. Lesser and D. D. Corkill, "The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks," *AI Magazine*, vol. 4, no. 3, pp. 15–33, 1983.

[5] H. V. D. Parunak, "Go to the ant: Engineering principles from natural multi-agent systems," *Annals of Operations Research, Special Issue on Artificial Intelligence and Management Science*, vol. 75, pp. 69–101, 1997.

[6] ——, "Generation and analysis of multiple futures with swarming agents," in *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10-14, 2010, Volume 1*, W. van der Hoek, G. A. Kaminka, Y. Lespérance, M. Luck, and S. Sen, Eds. IFAAMAS, 2010, pp. 1549–1550.

[7] M. Resnick, *Turtles, termites, and traffic jams: explorations in massively parallel microworlds*. Cambridge, MA, USA: MIT Press, 1994.

[8] K. Kelly, *Out of control: the rise of neo-biological civilization*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994.

[9] A. K. Engel, P. Fries, and W. Singer, "Dynamic predictions: oscillations and synchrony in top-down processing." *Nature reviews. Neuroscience*, vol. 2, no. 10, pp. 704–716, October 2001.

[10] D. E. Rumelhart, J. L. McClelland, and the PDP research group., Eds., *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundations*. MIT Press, 1986.

[11] F. Gruau, C. Eisenbeis, and L. Maignan, "The foundation of self-developing blob machines for spatial computing," *physica D:Nonlinear Phenomena*, vol. 237, 2008.

[12] B. Nitzberg and V. Lo, "Distributed shared memory: a survey of issues and algorithms," *Computer*, vol. 24, no. 8, pp. 52 –60, aug. 1991.

[13] P. M. B. Vitányi, "Locality, communication, and interconnect length in multicomputers," *SIAM Journal on Computing*, vol. 17, no. 4, pp. 659–672, Aug. 1988.

[14] H. Abelson, D.Allen, D. Coore, C. Hanson, G. Homsy, J. T. F. Knight, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss, "Amorphous computing," *Commun. ACM*, vol. 43, no. 5, pp. 74–82, 2000.

[15] E. Rauch, "Discrete, amorphous physical models," *International Journal of Theoretical Physics*, vol. 42, no. 2, pp. 329–348, feb 2003.

[16] J. Li and M. C. Chen, "Index domain alignment: Minimizing cost of cross-referencing between distributed arrays," in *Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, Md., Oct. 1990, pp. 424–433.

[17] J. Ghosh and K. Hwang, "Critical issues in mapping neural networks on message-passing multicomputers," in *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 3–11.

[18] K. Shahookar and P. Mazumder, "VLSI cell placement techniques," *ACM Computing Surveys*, vol. 23, no. 2, p. 143, Jun. 1991.

[19] L. Maignan and F. Gruau, "A 1D cellular automaton that moves particles until regular spatial placement," *Parallel Processing Letters*, vol. 19, no. 2, pp. 315–331, 2009.

[20] F. Gruau and L. Maignan, "Homogeneization of particles on a 2d hexagonal CA," 2010. [Online]. Available: http://blob.lri.fr/animation/blob.htm