MCMAS: A toolkit for developing agent-based simulations on many-core architectures

Guillaume Laville, Christophe Lang, Bénédicte Herrmann, Laurent Philippe, Kamel Mazouzi, Nicolas Marilleau

MCMAS: a toolkit for developing agent-based simulations on many-core architectures

Guillaume Laville* Christophe Lang* Bénédicte Herrmann*
Laurent Philippe* Kamel Mazouzi † Nicolas Marilleau‡

Abstract

Multi-agent models and simulations are used to describe complex systems in domains such as biological, geographical or ecological sciences. The increasing model complexity results in a growing need for computing resources and motivates the use of new architectures such as multi-cores and many-cores. Using them efficiently however remains a challenge in many models as it requires adaptations tailored to each program, using low-level code and libraries. In this paper we present MCMAS a generic toolkit allowing an efficient use of many-core architectures through already defined data structures and kernels. The toolkit provides few famous algorithms as diffusion, path-finding or population dynamics that are frequently used in an agent based models. For further needs, MCMAS is based on a flexible architecture that can easily be enriched by new algorithms thanks to development features. The use of the library is illustrated with three models and their performance analysis.

Keywords: Multi-Agent Systems, Parallel Computing, GPGPU, Many-core

1 Introduction

Multi-Agent Systems (MAS) are often used to describe large complex systems where the behaviour of the simulated entities cannot be generalised by a global law, as a mathematical differential equation for instance. This is the case in numerous biological, geographical or ecological systems [5, 8] where the behaviour of the entities composing the system can be represented by an algorithm, the agent algorithm. The behaviour observed on the whole system then emerges from the animated model.

In these simulations increasing the size or precision of the model is often needed to obtain more accurate or realistic results. This generally leads to a

*FEMTO-ST Institute, CNRS / Université de Franche-Comté, France
†Mésocentre de calcul de Franche-Comté, Université de Franche-Comté, France
‡UMI 209 - UMMISCO, Institut de Recherche pour le Développement (IRD), UPMC, France

higher computation load as more entities are represented or as more complex algorithms are used to simulate the entity behaviours. When the load increases personal computers may not be able to run the simulations in reasonable time and more powerful computing platforms must be used. Parallel architectures are thus becoming a required mean to gain performance. Their use however requires fundamental improvement in the model runtimes provided by standard platforms such as NetLogo [26], GAMA [28] or Repast [10]. Several projects as D-MASON [7] or Repast-HPC [11] have introduced distributed computing techniques into MAS in order to benefit from the parallel CPU architectures. The goal of these works is to accelerate or enlarge the simulations to get more descriptive and accurate results.

Last years have seen the emergence of GPU cards based on massively parallel architectures and many-core processors as the Xeon Phi, both used with a parallel SIMD (Single Instruction Multiple Data) programming model. GPGPU computing is already used in various domains as linear solvers, video streaming or image processing. Most of these applications are based on matrix data structures well-adapted to parallel processing and several general purpose libraries for high performance computing as BLAS [2] have already been adapted to these architectures. The underlying execution model, with lots of cores, well fits programs where several identical instructions are applied on large sets of data (SIMD model). This is, for instance, the case when a linear transformation is applied on a matrix. Many-core computing, i.e. based on many-core processors as the Xeon Phi, is an emerging domain that relays on more general purpose cores. The number of available cores is usually less than on GPU but their individual power is higher and they can run more general purpose applications using the OpenMP parallelism model.

MAS are also characterised by a set of entities having the same behaviour and should thus benefit from the use of many-core and GPU architectures. Considering that most of personal computers are equipped with an easily accessible GPU card, using GPU to run MAS must then be considered as a possible way to speed agent based simulations up. Similarly the emergence of many-core processors must be considered as an opportunity to improve agent based simulation performance. In practice MAS may however be characterised by not so regular data accesses and unpredictable behaviours due to algorithms offering multiple execution branches or random aspects. They thus do not perfectly fit the single instruction flow model and adaptations in their execution workflows are thus required to allow agents to run concurrently.

In this paper we assess the use of GPU and many-core architectures to accelerate MAS simulations. The contribution of the paper is a toolkit, called MCMAS (Many Core MAS), that provides functions to facilitate the implementation of MAS simulations on GPU and many-core architectures and better exploit their computing power. Related works on GPU and MAS are summarised in section 2. Then, in section 3, we present the MCMAS library. Its interface and extension facilities are illustrated on three use cases: the classical Prey-Predator model, the Collembola model and the Mior model. The model performance results are detailed in section 5. We then conclude on the possibilities of the MCMAS

platform and its possible application to other models or use cases in section 6.

2 Related works

Multi-agent systems (MAS) are produced by a bottom-up modelling approach, where the global evolution of the system is determined by the action of individual entities. These agents are associated to their own properties and behaviours, described in the shape of algorithm. Multi-agent systems cover a large spectrum of models, ranging from simple models implemented as cellular automaton to problems including state-of-art research in artificial intelligence. A distinction is thus made between cognitive and reactive agents [30, 9], depending on the level of reasoning associated to each individual. Cognitive agents characterised by complex behaviour are clearly incompatible with the SIMD execution model of the GPU. On the opposite, the behaviour of a reactive agents is only based on stimulus and reactions. These agents are used for instance to describe simple animals [23] or artificial creatures [21] bounded by a set of characteristics and definitions. As the reactive agents are characterised by more simple algorithms, run by all the agents, they are possible candidates to a GPU parallelisation.

MAS simulations usually relay on the scheduling at each time step of the execution of a set of agents having the same behaviour, hence the same algorithm. When the number of agents in the set becomes large it is worth to use more powerful computing architectures as GPU or many-core processors[1]. Some works have already demonstrated the gain of using GPU to run MAS in different applications as traffic simulation [27] or in various domains as cellular automaton [12], mobile agent path finding [14], genetic processes [22] or life science[17]. These works present specific adaptations of existing MAS to GPU. In these models individual behaviours driven by mathematical laws (path finding) or equations (cellular automatons) can be considered as the application of the same process on each individual. This approach does not however work for the majority of MAS, and algorithmic adaptations are often required. Note that a full-GPU approach sometime limits the possible use in MAS and that an hybrid approach as presented in [24], based on CPU plus GPU, may fit the needs of a larger number of MAS.

The FLAME-GPU [25] platform proposes an all-in-one solution to run MAS on GPU. The framework relies on a detailed XML description of the agents and on C-like code fragments to support several target architectures. This approach implies that the MAS is developed for the framework and thus cannot (re)use an existing model nor cannot interface with other MAS runtimes.

To our knowledge there is yet no work that explores the use of many-core processors (in the sense of many general purpose cores) to run agent based

[1] In the literature the term many-core is differently used either for GPU and many general prupose core processors (as the Xeon Phi) or just for many general prupose core processors to distinguish them from GPU. In the remainder of the paper we use multi-core for general purpose processors, with usually less than 10 cores, many-core for many-core processors as the Phi and GPU for GPU cards.

simulations.

3 MCMAS

As previously stated on the one hand Multi Agent Systems are based on large sets of entities with the same behaviour, and thus should benefit from a SIMD execution architecture, and on the other hand this behaviour is not always regular or simple enough to reach a full use of the power of GPU cards or many-core processors. For this reason, porting a MAS on these architectures often requires a model adaptation to benefit from this architecture. Many-core programming is however not as simple as developing models using MAS platforms. First, because the programming languages are not the same. Second, because there is not much facilities for the development or adaptation of models on such architectures. Note that GPU cards and many-core processors can also been used as co-processors to accelerate costly functions of a simulation as globally updating of an environment or a set of agents and we thus also explore this approach.

3.1 Basic MCMAS

We propose the MCMAS platform to facilitate the implementation of agent based models, or part of these models, on GPU and many-core architectures. MCMAS proposes a framework that includes a set of commonly used functions and data structures to simplify the implementation of new models and it allows the integration of new functionalities in the shape of plugins. Several levels of use are thus possible to provide a better flexibility in the integration between CPU and GPU execution.

Choosing a programming language is the first step to adapt an agent based model to a GPU or many-core platform. On the one hand the Java programming language is often used for the implementation of MAS due to its large availability and its high-level object-oriented programming. On the other hand GPU platforms only offer dedicated languages, CUDA or OpenCL, so that a model implementation requires some part of GPU-specific developments or the use of GPU enabled libraries. Many-core platforms may be programmed with standard languages but high performance languages as Open-MP, or OpenCL, are needed to efficiently benefit from their computing power. As Java is a widespread language in MAS, MCMAS offers a higher level Java interface. This interface is linked with OpenCL GPU-code through the JOCL library [1] (see figure 1). We choose OpenCL for its portability and the possibility to run programs on CPU, GPU and many-core architectures without modification.

As shown on figure 1 the framework provides two levels of interfaces. A low level interface MCM, for Many-Core Manager, that relays on the JOCL adaptation layer. MCM provides tools to manage the execution on several cores and to manage native memory. Plugins can be developed thanks to this low-level layer. These plugins are then grouped to provide the higher-level MCMAS

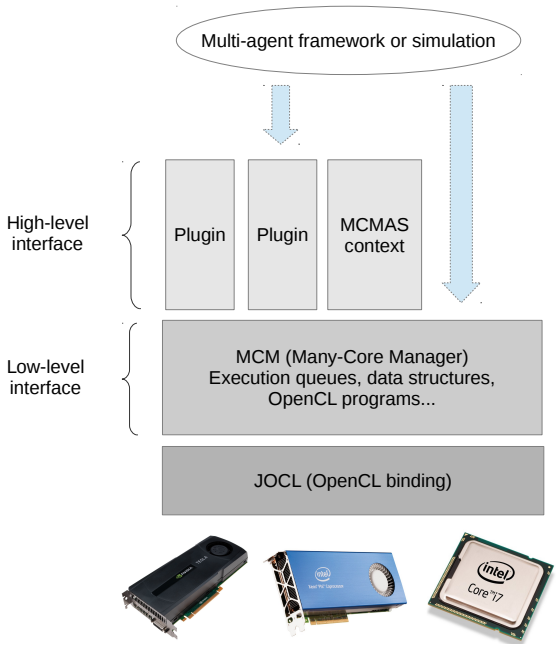


Figure 1: MCMAS general architecture

interface. Plugins are sets of functions used for similar problems (e.g. apply an operation on a whole grid). To facilitate the addition of new functions MCMAS is based on a dynamic architecture allowing the registration of new plugins at runtime.

The proposed architecture allows several parallelisation approaches depending on the model: (i) a **process parallelisation** where calls to plugins are issued only for the usage of many-core optimised primitives in an existing MAS model while the rest of the simulation still uses the CPU, or (ii) **model parallelisation** when the whole MAS model is rewritten to run on a parallel platform. This two layered extensible architecture allows the designer to either use already developed plugins or to roll its own solutions, based on the parallelised operations required by its model. Choosing either solution depends on the model analysis with regards to the Amdahl law [3], to identify the functions that would most benefit to run on several cores and their implementation difficulty. The process parallelisation is illustrated by the well known Prey-Predator model[20, 29]. The model parallelisation is illustrated by the Mior model[18]. Note that a third level of use is also possible when an existing plugin can be adapted to fit the needs of a model. This third way of using MCMAS is illustrated by the Collembola model. These model parallelisation are developed in section 4.

3.2 Programming with MCMAS

OpenCL provides access to CPU, GPU or many-core threads using an asynchronous interface. This library is based around three main concepts. The *kernel* represents a program to be executed on the GPU or many-core. The *work-item* is analogous to the concept of thread on CPU. The *work-group* is a set of work-items that share memory. An OpenCL execution consists in running the same kernel on numerous work-items. Synchronisation operations, as barriers, can only be used across the same work-group. Data used by the work-items can be stored in local (high speed) or global (low speed) GPU or many-core memory. Since the size of this local memory is often limited to a few hundred of kilobytes, choosing this number often implies a compromise between the model synchronisation or data requirements and the available resources. In the case of agent based simulations, each agent can thus naturally be mapped to a work-item. Work-groups can then be used to represent groups of agents or simulations sharing common data (as the environment) or algorithms (as the background evolution process). This process is described in more details in the case of the Mior model[18].

Similar data structures are used by whole classes of MAS. One such example is the grid, which can be either integrated in the algorithm, as in SugarScape [12] where each cell represents the fundamental unit of modelling, or used to discretise a continuous environment as path-finding simulations [15]. These grids can be considered as 2d or 3d matrices representing agent data or their environment. Another data representation often encountered in MAS is the usage of coordinate systems to position agents in the simulation space [13].

MCMAS provides a standard set of data structures (grids, vectors, structures) as a mean to pass data to plugin methods. Methods are provided to facilitate the translation of existing Java data structures to these formats. Each plugin is also free to define its own data structures, either for its own usage or for general use. A flexible data architecture has been designed to expand the existing data collection.

These common structures also lead to the usage of similar algorithms in many simulations, such as distance computation in 2d or 3d space, diffusion processes, reduction operations, SIMD transformations (as linear functions) applied to each cell of the model. These kinds of processing can be parallelised and executed on the GPU, leading to possible performance gains, without heavy modifications to the model scheme.

The MCMAS interface is based on the `MCMASContext` object. This context contains all the data (for instance device context and execution queue) required to run MCMAS low-level operations and plugins. The context can be created using a wide variety of constructors, to allow the customisation of the environment depending on the available execution resources and the needed functionalities: profiling, debugging... This MCMAS context can then be either used to initialise MCMAS plugins using the `newInstance()` method, or to directly call low-level OpenCL operations, using the accessors provided for the underlying OpenCL objects.

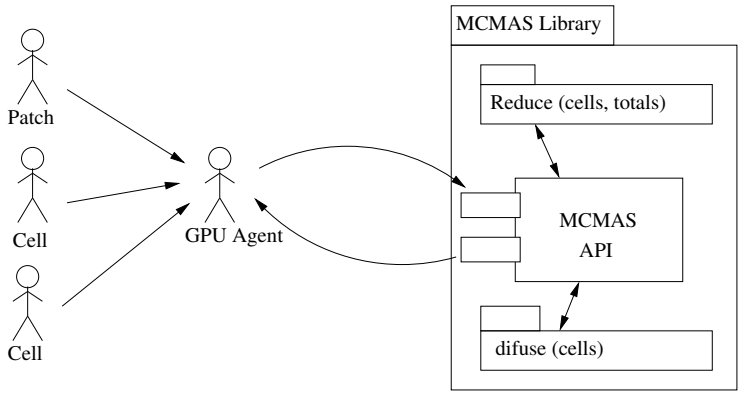


Figure 2: MCMAS integration in MAS framework

3.3 Developing a new MCMAS Plugin

A MCMAS plugin is a Java class which implements the `MCMASPlugin` interface. This interface allows plugins to be instantiated from a MCMAS context. This context is then be used by the plugin to allocate new data structures, to launch operations or to adapt its execution.

Beyond the basic methods required by the interface, each plugin can provide its own free-form set of operations. New MCMAS plugins and structures are packaged as Java libraries. By convention all classes belonging to the same plugin are located in a `mcmas.plugins.<plugin name>` package to maintain code isolation and facilitate the discovery of new plugins.

Most plugin-provided operations are organised around the standard GPU execution workflow: (1) OpenCL source code retrieval and compilation, (2) copy of Java data into input data structures, (3) execution of one or more kernel, (4) retrieval of output data and translation into Java data structures, and (5) resource cleanup and return from the primitive call. Memory allocations in each plugin can be managed at two levels. At instance level the memory is used for the lifetime of the plugin. At method level the memory is used for temporary copies of input and output parameters and to manage the execution progress.

3.4 Using MCMAS from existing MAS frameworks

The MCMAS library can also be used to delegate computations in existing MAS frameworks. The library can be interfaced with existing agent platforms to acts as a wrapper for OpenCL code that allows integration of optimised model parts within other simulators. An intermediary agent is required to translate the requests issued from existing models into calls to MCMAS plugins, and to manage the interactions with the MCMAS platform as shown on figure 2. This translation layer between MCMAS and the MAS framework allows a transparent use of the library without disturbing the existing model architecture. For instance, in

the case of the GAMA platform, this integration can be realised by extending the agent description language with MCMAS related functions. For that, an OGSI plugin dedicated to GAMA can be developed. In the case of the Madkit framework [16], a threaded agent may be implemented that is ordered by other simulation agents through a dedicated agent communication language.

4 Parallelising models with MCMAS

In this section we present three examples of model parallelisation. Each case illustrates a possible use of MCMAS, i.e. using existing plugins (process parallelisation), adapting existing plugins to accelerate part of a model (process parallelisaton with plugin adaptation) or developing a model to run on several cores (model parallelisation).

4.1 Process parallelisation use case

As a first illustration of the MCMAS use we present here an adaptation of the well known Prey-Predator model. The observed system consists of wolves, sheeps and grass. Its implementation is based on five steps that are run sequentially: environment preparation (conversion of the environment grid from the java representation to the MCMAS representation), grass growth, preparing agent position (conversion of agent positions from their java representation to their MCMAS representation), maximum search to define the agent moves depending on their neighbourhood and eventually the agent updates (moves, feeding, reproduction, energy). Two of these steps can easily be run on many cores as the needed functions (linear transformation of all the cells of the grid and maximum search) are already implemented in MCMAS. The other parts of the model are harder to implement on several cores due to synchronisation issues and conditional behaviours.

Figure 3 illustrates the use of MCMAS to speedup the Prey-Predator model. This process parallelisation is done at very low cost as we just call plugins. It is also easy to realise for none expert programmers. The main loop of the model is run on the CPU. This means that the CPU keeps the control of the simulation run. The two most costly functions, grass updating and maximum search, are run in parallel on the GPU or the many-core processor, as on a co-processor. To illustrate the MCMAS use for a process parallelisation, a scheme of the simulation program is given on figure 4.

This usage of the MCMAS library is rather easy to integrate in an existing model. It however assumes that there always exists an adapted plugin for the more costly steps of the model. This is fortunately the case for the Prey-Predator model but this is obviously not the case for all models. For this we provide other implementation levels in MCMAS that allows the developer to adapt existing plugins to develop new ones.

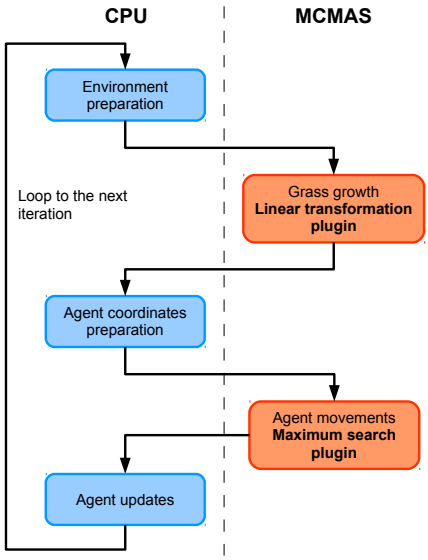


Figure 3: Prey-Predator implementation over MCMAS

```
1  public void run() {
2      MCMASContext context = new MCMASContext(MCMAS.GPU);
3      LinearPlugin linPlug = LinearPlugin.newInstance(context);
4      SearchPlugin searchPlug = Searchlugin.newInstance(context);
5      for nbIter {
6        envGrid = doEnvPreparation(..);      // prepare environment on CPU
7        linPlug.transfrom(envGrid)           // grass growth on GPU
8        agentVect = doAgentCoordonate(..);   // prepare agent coordinates on CPU
9        Searchlugin.searchMax(agentVect);    // Max search on GPU
10       agentUpdate(agentVect);              // Agent updates on CPU
11     }
12 }
```

Figure 4: Usage of MCMAS in the Prey-Predator model

4.2 Process parallelisation with plugin adaptation

The Collembola model is focusing on landscape biodiversity. It reproduces the diffusion of arthropods life forms (collembola) across plots of an identified territory. This environment is constituted of forests, cultivated and artificial areas. The model goal is to study the impact of modifications of this environment on biodiversity. The model subdivides each plot in surface units and follows the evolution of the number of individuals in time. The model is thus based on a grid of cells (the surface units) that are grouped in plots. Its evolution can be decomposed into four steps: (i) *sum* of individuals, which mainly consists in summing the collembola in each plot cell by cell, (ii) *reproduction* of each population in parcels, this consists in creating new individuals in each plot depending on the previously computed sum, these individuals are then distributed in the

cells, (iii) *diffusion* between cells, which consists in computing the population gradient and applying the overflows (individuals that migrate from one cell to another) and (iv) *selection* of surviving individuals, which consists in applying a boolean condition on each cell.

A first analysis of the problem shows that a simple way to improve the simulation performance with many cores is to use existing plugins to compute each of the four steps. An existing plugins can be used for the *selection* step as it is applied to all the cells. The computations done by the three other steps are close to functions proposed by existing plugins, but they depend on the plot configurations. For this reason we had to adapt them to the particular Collembola case. The *sum* step sums the collembola number for each plot instead of for the whole grid so we have modified the plugin to return a vector instead of a single value. The *reproduction* step uses the sum of the collembola in the plot to determine the number of created collembola, then it distributes them in the cells under a maximum threshold condition. This has been implemented using the linear transformation plugin that we adapt to these constraints. The *diffusion* step needs to compute a gradient of diffusion between the cells under the condition of the plot type. We have modified the gradient plugin to add this condition.

Note that specific synchronisation schemes are used with these four steps to overlap the data transfers between the host memory and the GPU or many-core memory.

Thanks to the MCMAS architecture the modified plugins are easily added to the framework. The resulting code is thus quite similar to the one presented for the Prey-Predator model. We just call the four plugins instead of two. As previously the GPU or the many-core platform is used as a specialised co-processor for the simulation.

4.3 Model parallelisation use case

The Mior (MIcro-ORganism) [6] model simulates local interactions in a soil between microbial colonies and organic matters. The Mior model can be used in multi-scale MAS, such as Sworm [4]. Since the evolution takes place at a microscopic scale each unit of soil corresponds to many such simulations that justify the computing cost of this process.

```
1 // Create a new MIOR model template
2 MiorWorld model = new MiorModel();
3 model.nbOM = 310; model.nbMM = 38; model.size = 200;
4 MCMASContext context = new MCMASContext(MCMASContext.GPU);
5 MiorPlugin plugin = MiorPlugin.newInstance(context);
6 // Execute 100 instances simulating 1000 steps each time.
7 int [][] CO2Values = new int[100][1000];
8 plugin.runNSimulations(model, 100, CO2Values, 1000);
```

Figure 5: Usage of the Mior plugin from Java code

The Mior model is based on two types of agents: (i) the Meta-Mior (MM),

```
1  public class MiorPlugin extends MCMASPlugin<MiorPlugin> {
2    // Static method implemented by all MCMAS plugins (factory)
3    public static MiorPlugin getInstance(MCMASContext context) {
4      new MiorPlugin(context.getContext(), context.getQueue();
5    }
6    private MiorPlugin(Context context, CommandQueue queue) {
7    }
8    public void runNSimulations(...) {The
9  model is based on a grid of cells (the surface units) that are grouped
10 in plots. The arrival step mainly consists in counting the collembolas
11 in
12   }
13 }
```

Figure 6: Implementation of the Mior plugin

microbial colonies consuming carbon and (ii) the Organic Matter (OM), carbon deposits occurring in soil. Each Meta-Mior agent exhibits two distinct behaviours. By *breathing* it converts mineral carbon from the soil to carbon dioxide CO_2, released in the soil. By *growing* it fixes the carbon present in the environment to reproduce itself (augments its size). The *growing* action is only possible if the colony breathing needs are covered, i.e. if enough mineral carbon is available.

The Mior model is not based on a grid and many of its computations cannot be found in a generic plugin so we have developed a specific MCMAS plugin. Figure 6 gives a scheme of the MCMAS implementation for the plugin with specific functions. Once this plugin is developed it can easily been used from a Java program as illustrated on (Figure 5). Implementing a new plugin however requires specific many-core programming and parallelisation skills to efficiently use the resources. For instance the Mior plugin uses compressed data structures to improve the execution performance. The Mior plugin makes use of the MCM interface. It facilitates its implementation by providing functions for many-core run and memory management.

5 Experiments

In this section we present the performance obtained with the implementation of the three use cases. We show the gain obtained by delegating functions to MCMAS or by implementing a plugin that uses the MCMAS library. With these experiments we illustrate the possible uses of MCMAS depending on the considered GPU.

5.1 Experiment settings

There are lots of GPU cards sold by manufacturers and it is obviously not possible to test MCMAS on all of them. As we can only test few of them we had to choice cards that are representative of different categories.

One of the MCMAS targets is the improvement of the simulation performance on personal computers. We thus choose to first test our framework on mainstream cards. We choose two cards: one Nvidia card, the Geforce 560Ti, and one AMD card, the Radeon HD HD6870. Both can be considered as representative of actual cards albeit they are not the most up-to-date. They both have 1 Gb of dedicated memory. These cards are included in two different personal computers. The Geforce card has 384 cores running at a frequency speed of 833 MHz. It is associated with an Intel core i7 2600K processor, with 4 physical cores, running at 3.4 Ghz and 4 Go of central memory. The Radeon card has 1120 cores running at a frequency speed of 900 MHz. It is associated with an AMD Phenom II X6 1090T running at 3.2 GHz, with 6 physical cores, and 4 Gb of central memory. These configurations are standard configurations for todays personal computers.

MCMAS can also been used on supercomputers. As more and more clusters are equiped with GPU cards to improve their computing power, it is worth testing MCMAS with professional cards. Our tests are based on two generations of Nvidia GPU cards: the Kepler K20M and K40. The K20 card has 2494 cores running at a frequency clock of 706 MHz and 5 Gb of dedicated memory. The K20 card has 2880 cores running at 745 MHz and 12 Gb of memory. They are associated with an Intel Xeon E5-2609v2 running at 2.5 GHz. This card is a recent GPU card mostly designed for high Performance Computing (HPC). We did also run the simulations on a Nvidia Tesla C1060 card. This is a rather old card launched in 2008. This card gives good results for dedicated code as the Mior plugin but lakes of genericity for less specific implementations, mainly because it does not provide cached memory. The results with this older card are presented in [19]. Last, to illustrate the many-core performance, we also run our models on a Xeon Phi card. The Xeon Phi processor has 61 physical cores that provides 244 virtual cores running at 1.238GHz with 1 Gb of memory.

Using an accelerator cards as a GPU or a Xeon Phi always results in an additional cost, at least the cost of transferring data between the CPU and the card memory. These costs can be minimised when the size of the simulation increases and the performance gains thus depends more on the size of the model. Therefore we have run all the presented models with different scaling factors for the agent population size. On the other hand, running small simulations on the external card may not be a good idea for particular models due to these overhead of memory transfers and it is not worth it implementing a many-core simulation if its size does not justify it. Despite this we did not notice a performance degradation on our models, even for small sized simulations, as it can be noticed on the presented performance curves.

For each simulation we measure the running time on the CPU and on the accelerator card and compare both performance. The CPU runs are Java implementations of the models instead of their initial Netlogo implementations as it would not be fair to compare programs to interpreted code (Netlogo implementations are much more slower). The basic CPU runs are sequential runs. Using the ability of OpenCL, and hence MCMAS, to run on multi-core CPU as well as on GPU and many-core, we also includes multi-core CPU runs of the

models. This gives a comparison of the performance obtained on the CPU by using all the available cores, since MCMAS also targets multi-core platforms.

With these experiments we intend to show on the one hand how MCMAS can improve simulations by simply using or adapting existing plugins and, on the other hand, how it can also be used to develop dedicated plugins and give good performances on personal computers as well as on HPC platforms. The first set of experiments only presents the results on personal computer platforms. Then, in a second set, we show the results obtained on HPC platforms.

5.2 Process parallelisation results

Our first illustrating model is the Prey-Predator model, a case of simple process parallelisation. The model is implemented as presented in section 4.1 with two functions delegated to MCMAS and therefore to the GPU. The objective of this experiment is to show how a GPU card of a personal computer and MCMAS can improve the simulation performance in case of a simple adaptation of the model (call to a dedicated library). The presented implementation of the Prey-Predator model is probably not the most efficient implementation of the model. It just intends to be an illustration of how MCMAS can speed up a model at low programming cost and complexity.

With this model we assess two cases where the use of more computing power is needed: increasing the size of the model and varying one of the simulation parameters, the size of the search area here. The first case shows how, depending on the scaling factor, the whole simulation is impacted by the GPU use. In the second case only the search plugin is impacted by the parameter variation and the remaining of the simulation is not changed.

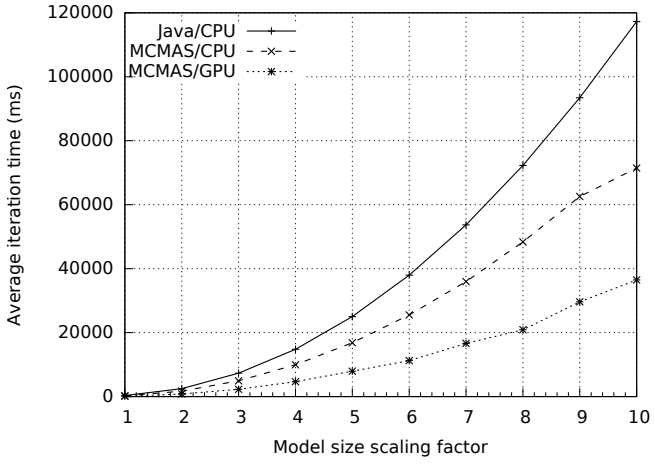


Figure 7: Prey-Predator, performance depending on environment size

Figure 7 shows the performance obtained on the GeForce platform with the Prey-Predator model when varying the size of the model. Three implementa-

tions are compared here: (1) the Java/CPU curve plots the performance of a traditional sequential implementation of the model, (2) the MCMAS/CPU curve plots a parallel implementation that runs on the 8 cores (4 physical cores) of the CPU, and (3) the MCMAS/GPU curve plots the GPU runs of the same implementation. Each point of the curves is the mean running time of 5 runs[2]. Note that the performance obtained by the two parallel implementations are prone to variation, around 15% between the extreme values. This variation increases with the model size and is due to concurrency in memory accesses. In the initial simulation (scale 1) the environment is a 500×500 grid with 10000 preys and 5000 predators randomly disseminated on the grid. Note that a scale of two between two simulations causes a quadratic increase of the number of agents to maintain the agent density on the grid. For these simulations the size of the search area is fixed at 50. The figure shows that using the GPU to delegate the computation of some functions in the simulation may leads to significant performance improvements, globally 4 times compared to the Java implementation and 2 times compared to the CPU run whatever the model size is.

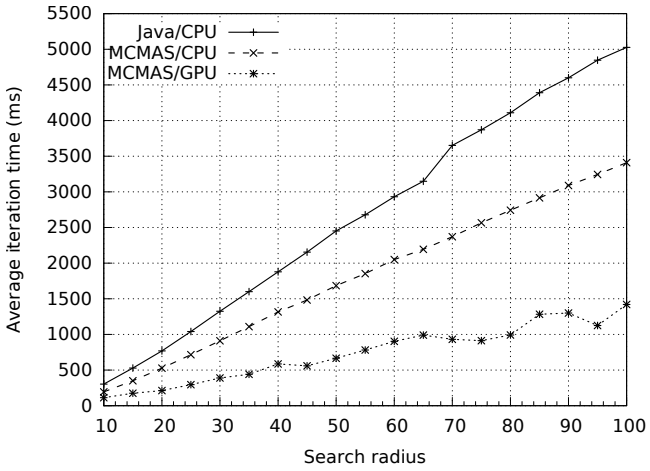


Figure 8: Prey-Predator, performance depending on the search area size

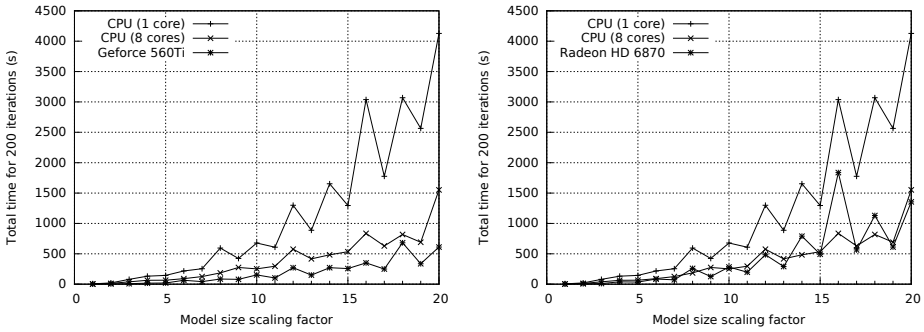
Figure 8 shows the results obtained by the Prey-Predator model when varying the size of the search area on the GeForce GPU card. The variation thus only impacts the search plugin runtime while keeping the same grid size and the same running time for the other parts of the code. This illustrates that one of the parameter of the simulation may have a significant impact on the simulation and its parallelisation. The simulation used is the reference simulation (scale 1) used in Figure 7. The search area is defined by a square which side length is twice the search radius. The size of the search area has thus a quadratic progression when the search radius linearly progress.

[2]We did 7 runs for each measure and removed the two extreme values to improve the quality of the data set

On this simple model, by just using existing MCMAS plugins to improve costly parts of the simulation (as shown on figure 3), we can speedup the global runs up to 3.3 times faster compared to the sequential CPU runs with the GeForce platform and 2.3 times faster than when using the cores of the CPU. Variations can be observed on the GPU curves that are mainly linked to granularity of the simulation compared to the cache size as on the previous figure.

5.3 Process parallelisation with plugin adaptation results

The Collembola model illustrates a case of process parallelisation with plugin adaptation. The model is tested on both the GeForce and the Radeon cards. We did not implement the Collembola model on Java and we just compare here the impact of increasing the number of CPU cores used to run the simulation to the GPU runs.



(a) GeForce GPU card (b) Radeon GPU card

Figure 9: Collembola: performance depending on the size of the simulation

Figures 9a and 9b show the results obtained when running the simulations on both the Radeon card and the GeForce card. We compare here a sequential java run (1 core) and a multi-core MCMAS run on the local CPU (8 cores) to a GPU run (GeForce or Radeon). Each dot on the curves is the mean value of 5 runs[3] of the simulation. The size of the reference simulation (scaling factor of 1) is 256×256. The scale is obtained by multiplying the size of the environment while maintaining the global collembola density. Increasing the scaling factor thus leads to a quadratic progression of the observed area size.

As expected the use of the MCMAS library increases the performance of the simulation compared to the sequential runs. The best observed speedup is factor of 8 between the sequential run and the GeForce run. Note that the Radeon run gives poorer performance, limited to a speedup of 2.6. The reason is that the Radeon card is older and less multi-purpose and thus does not give so much

[3]We did 7 runs for each measure and removed the two extreme values to improve the quality of the data distribution

performance as the agent code is not regular enough. Similarly the multi-core CPU curves are very close to the GPU ones and we have noticed a rather light load of the GPU during the simulations which means that our code does not take benefit of the GPU full power has it does not perfectly match the SIMD programming model. This enforces our initial assumption that GPU platforms must gain in flexibility to be used with MAS, which is fortunately the case for newer platforms. The curves exhibit a odd-even pattern, more marked for the Radeon. Since this phenomenon is visible on distinct hardware, drivers and OpenCL implementations, it is likely due to the model decomposition process based on warp of fixed power-of-two sizes.

5.4 Model parallelisation results

The Mior model illustrates the implementation of a complete model in the shape of a plugin. The Mior experiments illustrate the impact of increasing the level of adaptation of the algorithm to GPU hardware. Parallelising a model and adapting it to the GPU architecture may indeed be a complex work and several iterations are sometime necessary to obtain good performance. For this reason, we show here three successive versions of the Mior implementation. The GPU v1.0 Mior implementation is a direct implementation of the existing algorithm and its data structures. The GPU v2.0 Mior implementation uses compact representations of the topology provided by the low level MCM interface. The GPU v3.0 Mior implementation tries to benefit from the local memory by explicitly copying the most used data at the end of each computations.

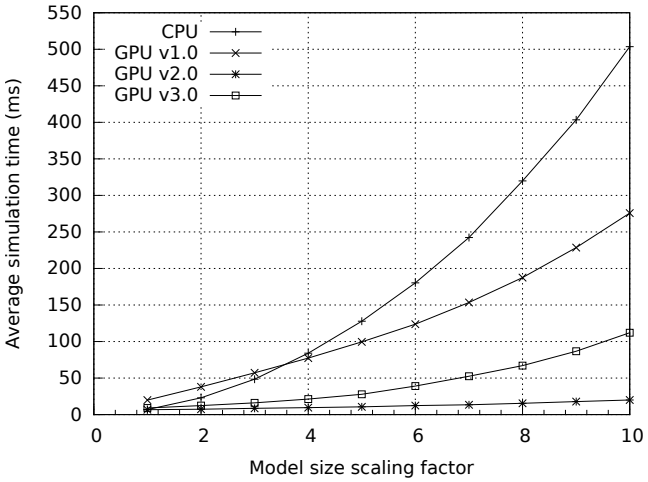


Figure 10: Mior performance on mainstream GPU

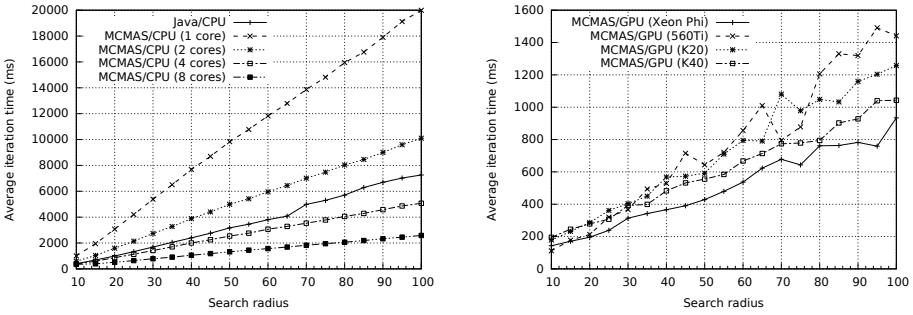
Figure 10 gives a performance comparison of the simulation runs on the GeForce GPU and its associated CPU. Each dot represents the mean value of

the execution duration for 50 simulations. At scale 1 the model contains 38 MM and 310 OM which is dictated by the original simulation. Then, at each scale, the size of the model, the number of OM, the number of MM and the size of the environment are multiplied by the corresponding factor to maintain the same mean agent density in the model.

We can notice that there is no notable benefit to run on GPU for scale 1. The GPU implementation does not have enough threads (representing agents) for an optimal usage of GPU resources. But from scale 2 to 10 the speedup increases from 2 to 25 if we consider the CPU and GPU v2.0 results. The speedup is much higher than for the previous models. This is due to the fact that the whole simulation is parallelised and adapted to the GPU constraints. In the previous models only some functions were delegated to the GPU while the main control structure still run on the CPU. From the GPU v1.0 curve we can however conclude that parallelising the model is not always that simple. This first version relays on the synchronisation tools provided by OpenCL and this is not efficient in this case with too much conflicts. On the other hand the GPU v3.0 is shown to be underperforming although it was developed to enhance the efficiency. In this case the memory copies do not provides the expected gain in performance. This clearly illustrates the difficulty to develop an optimal code on the GPU architecture.

5.5 Results on HPC platforms

We show here the results obtained on our HPC platforms presented in section 5.1. The experiments setting for each model are the same as for the personal computer experiments. We compare each model when running on the CPU of the computing node and on the GPU or Xeon Phi.



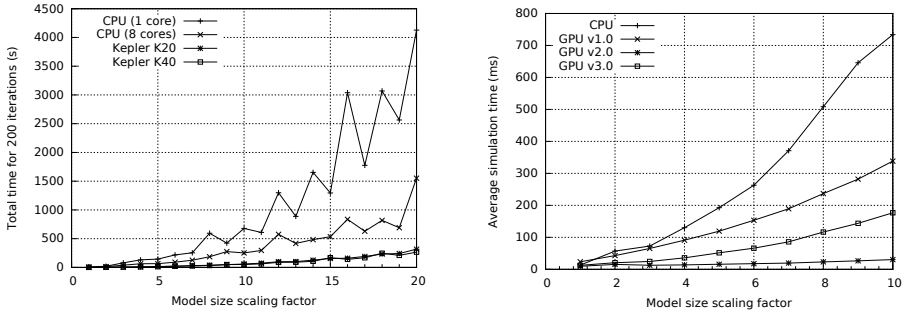
(a) Performance on multi-cores

(b) Performance on many-core and GPU

Figure 11: Performance for the Prey-Predator model on HPC platforms

Figures 11a and 11b show the performance of the Prey-Predator model when varying the search radius of the predators on several HPC platforms. Figure 11a shows the compared performance of the model for different CPU configurations.

For the reference curve we run the Java implementation of the model on the node CPU. Then we use the MCMAS implementation and run it with different numbers of cores. The 1-core run generates a significant lost of performance. This is explained by the initial cost of generating MCMAS data structures. Then the performance increases with the number of cores until it reaches a speedup factor of 4 for 8 cores. Figure 11b shows the performance results for different GPU and many-core platforms. The Xeon Phi provides the best performance but the Kepler GPU cards give very close results. The GeForce card has been added here for comparison and we can note that, although its performance is lower it is not that far for the HPC cards. When comparing both Figures 11a and 11b, we can see that the running time almost linearly increases with the search radius and that the performance of the K40 card and the Xeon Phi shows a speedup of 2 compared to a 8 cores CPU run and a speedup of 10 compared to the reference Java run.



(a) Collembola model

(b) Mior model

Figure 12: Performance of Collembola and Mior models on HPC platforms

Figures 12a and 12b shows the results obtained on Kepler GPU for the Collembola and Mior models. Figure 12a shows a comparison of the results for the two Kepler platforms and for the multi-core run. Here the K20 card reaches almost the same performance as the K40 card. This can be explain by the short difference between their frequencies and light load of the devices although the gain is noticeable. Note that the 8 cores runs give an average performance. Figure 12b shows the results for the different Mior implementations on the K40 card. The obtained performance is significantly better here than on the GeForce as we reach a factor of 40.

On these HPC results we see that the speedup varies between 10 (Prey-Predator and Collembola models) up to 40 (Mior model). As for the mainstream cards the difference is explained by the level of adaptation of the implementation to the GPU architecture. With a speedup of 40 we reach a rather standard value that can be reached when porting other applications on GPU, as linear algebra based applications for instance.

6 Conclusion

In this article we present MCMAS, a solution that facilitates the use of many-core architectures and that allows the integration of optimised model parts within agent based simulators. To achieve this, two possible approaches are supported by our toolkit: (i) use MCMAS as an optimised algorithm library; or (ii) use MCMAS as many-core runtime to develop specific algorithms or MAS. The usage can be mixed by the model designer, depending on its model needs and of the amount of development required.

The first approach is to use the interfaces and plugins already provided by MCMAS. These plugins cover classic problems in MAS simulations as path-finding, diffusion or population dynamics. These predefined algorithms are ready to be used for accelerating one or more parts of an existing CPU simulation, without huge changes in the existing implementation. Thanks to the dynamic architecture new plugins can be added at runtime and it is thus easy to modify standard plugins to adapt them to specific cases as illustrated with the Collembola model.

The second approach is to develop new plugins for MCMAS to enable the implementation of more specialised or performance-critical algorithms directly on the underlying many-cores platform. This approach implies the development of OpenCL kernels, called from MCMAS, to execute portions of the computations. Once written, these kernels can be used both on CPU, GPU, many-core processors or any other OpenCL supported platform: this allows to reuse the same program on a wide set of architectures, ranging from personal computers to computing clusters, or dedicated GPU nodes, without modification or manual re-compilation.

Our main goal is now to enrich the MCMAS platform to support more MAS problems and to refine the support of new data structures to generalise the possible applications of this platform. The platform is now freely available on our website[4] and can also be completed by other contributors.

Note that MCMAS has yet only be tested on four GPU cards, Xeon Phi and on few CPU cores of a standard CPU. It will interesting to test it on more GPU architectures or even FPGA based platforms. While simple runs should quickly be possible as MCMAS if OpenCL is available on these platforms, efficient use may leverage more challenges due the difference of their design.

7 Acknowledgements

Computations presented in this article were realised on the supercomputing facilities provided by the **Mésocentre de calcul de Franche-Comté**. The authors would like to thank M. Cédric Clerget for its help in running the simulations on the Radeon card.

[4] `https://disc.univ-fcomte.fr/gitlab/guillaume.laville/mcmas/tree/master`

References

[1] JOCL: Java bindings for OpenCL. `http://www.jocl.org/`. [24-nov-2014].

[2] The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS). `https://developer.nvidia.com/cublas`. [24-nov-2014].

[3] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[4] E. Blanchart, N. Marilleau, A. Drogoul, E. Perrier, JL. Chotte, and C. Cambier. Sworm: an agent-based model to simulate the effect of earthworms on soil structure. *European Journal of Soil Science*, 60(1):13–21, 2009.

[5] Frank J Bruggeman and Hans V Westerhoff. The nature of systems biology. *TRENDS in Microbiology*, 15(1):45–50, 2007.

[6] M. Bousso C. Cambier, D. Masse and E. Perrier. An offer versus demand modelling approach to assess the impact of micro-organisms spatio-temporal dynamics on soil organic matter decomposition rates. *Ecological Modelling*, 139(1-2):301–313, 2007.

[7] Michele Carillo, Gennaro Cordasco, Rosario De Chiara, Francesco Raia, Vittorio Scarano, and Flavio Serrapica. Enhancing the performances of d-mason - a motivating example. In Nuno Pina, Janusz Kacprzyk, and Mohammad S. Obaidat, editors, *SIMULTECH*, pages 137–143. SciTePress, 2012.

[8] Jean-Christophe Castella, Suan Pheng Kam, Dang Dinh Quang, Peter H. Verburg, and Chu Thai Hoanh. Combining top-down and bottom-up modelling approaches of land use/cover change to support public policies: Application to sustainable management of natural resources in northern Vietnam. *Land Use Policy*, 24(3):531 – 545, 2007. Integrated Assessment of the Land System: The Future of Land Use.

[9] Brahim Chaib-Draa, Imed Jarras, and Bernard Moulin. Systèmes multi-agents: principes généraux et applications. *Edition Hermès*, 2001.

[10] Nicholson Collier. RePast: An Extensible Framework for Agent Simulation. *Natural Resources and Environmental Issues*, 8(4):17–21, 2001.

[11] Nicholson Collier and Michael North. Parallel agent-based simulation with REPAST for high performance computing. *SIMULATION*, 2012.

[12] R. M. D'souza, M. Lysenko, and K. Rahmani. Sugarscape on steroids: Simulating over a million agents at interactive rates. In *Proceedings of the Agent 2007 Conference*, 2007.

[13] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient GPU implementation for large scale individual-based simulation of collective behavior. In *Proceedings of the 2009 Int. Workshop on High Performance Computational Systems Biology*, HIBI '09, pages 51–58, Washington, DC, USA, 2009. IEEE Computer Society.

[14] R. Silveira et.al. Path-planning for RTS games based on potential fields. In *Proceedings of the Third international conference on Motion in games*, MIG'10, pages 410–421, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] L. Fischer, R. Silveira, and L. Nedel. GPU accelerated path-planning for multi-agents in virtual environments. In *Proceedings of the 2009 VIII Brazilian Symposium on Games and Digital Entertainment*, SBGAMES '09, pages 101–110, Washington, DC, USA, 2009. IEEE Computer Society.

[16] O. Gutknecht and J. Ferber. Madkit: a generic multi-agent platform. In *Proceedings of the fourth international conference on Autonomous agents*, AGENTS '00, pages 78–79, New York, NY, USA, 2000. ACM.

[17] G. Laville, N. Marilleau C. Lang, K. Mazouzi, and L. Philippe. Using GPU for multi-agent soil simulation. In *PDP 2013*, pages 392–399, Belfast, Ireland, February 2013. IEEE Computer Society Press.

[18] G. Laville, K. Mazouzi, C. Lang, N. Marilleau, and L. Philippe. Using GPU for multi-agent multi-scale simulations. In *DCAI'12*, volume 151 of *Advances in Intelligent and Soft Computing*, pages 197–204, Salamanca, Spain, March 2012. Springer.

[19] Guillaume Laville, Kamel Mazouzi, Christophe Lang, Nicolas Marilleau, Bénédicte Herrmann, and Laurent Philippe. MCMAS: a toolkit to benefit from many-core architecture in agent-based simulation. In *PADAPS 2013, 1st Workshop on Parallel and Distributed Agent-Based Simulations, in conjunction with EuroPar 2013*, volume 8374 of *LNCS*, pages 544–554, Aachen, Germany, August 2013. Springer.

[20] A. J. Lotka. Analytical Note on Certain Rhythmic Relations in Organic Systems. *Proceedings of the National Academy of Sciences of the United States of America*, 6(7):410–415, July 1920.

[21] P. Maes. *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*. A Bradford book. MIT Press, 1990.

[22] O. Maitre, N. Lachiche, P. Clauss, L. Baumes, A. Corma, and P. Collet. Efficient parallel implementation of evolutionary algorithms on GPGPU cards. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *LNCS*, pages 974–985. Springer Berlin / Heidelberg, 2009.

[23] D. McFarland. *The Oxford companion to animal behaviour*. Oxford Paperback Reference. Oxford University Press, 1987.

[24] Fabien Michel. Gpu environmental delegation of agent perceptions for mabs. In Mohamed Essaaidi and Mohamed Nemiche, editors, *ICCS'12, IEEE International Conference on Complex Systems, Agadir, Morocco, November 5-6*, pages 1–6. IEEE Computer Society, 2012.

[25] Paul Richmond. FLAME GPU Technical Report and User Guide (CS-11-03). Technical report, Department of Computer Science, University of Sheffield, 2011.

[26] E. Sklar. Netlogo, a multi-agent simulation environment. *Artificial Life*, 13(3):303–311, 2011.

[27] D. Strippgen and K. Nagel. Multi-agent traffic simulation with cuda. *2009 International Conference on High Performance Computing Simulation*, pages 106–114, 2009.

[28] P. Taillandier, A. Drogoul, D.A. Vo, and E. Amouroux. Gama: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In *The 13th International Conference on Principles and Practices in Multi-Agent Systems*, volume 7057/2012, pages 242–258, India, 2012.

[29] Hugo Thierry, David Sheeren, Nicolas Marilleau, Nathalie Corson, Marion Amalric, and Claude Monteil. From the lotka-volterra model to a spatialised population-driven individual-based model. *Ecological Modeling, Elsevier*, to-appear, 2014.

[30] Eric Werner and Yves Demazeau. The design of multi-agent systems. *Decentralized AI*, 3:3–30, 1992.

8 Biographies of authors

Guillaume Laville obtained his Ph.D degree in 2014 at the University of Franche-Comté. He is currently a system engineer at the computing centre of Université de Franche-Comté. His research interests include distributed and parallel platforms, in particular many-core and GPU, and multi-agent systems.

Christophe Lang obtained his PhD Thesis in Computer Science (University of Franche-Comté) in 1999. He is an associate professor the Computer Science Department (DISC) of Institut FEMTO-ST at the Université of Franche-Comté (UFC) (France). His research interests focus on distributed systems, multi-agent systems, simulation, models and sensor networks.

Bénédicte Herrmann obtained his Ph.D degree in 1993 at the Université of Franche-Comté. She is currently an Assistant Professor in the Computer Science Department of Institut FEMTO-ST (DISC) at the Université of Franche-Comté (UFC) (France). Her main research interests include distributed systems and complex system modeling.

Laurent Philippe obtained his Ph.D degree in 1993 at the Université of Franche-Comté and his HDR in 2000. He is currently a full Professor in the Computer Science Department (DISC) of Institut FEMTO-ST at the Université of Franche-Comté (UFC) (France). His main research interests include distributed and parallel systems, scheduling algorithms for parallem platforms and optimization procedures for complex systems.

Kamel Mazouzi obtained his Ph.D degree in 2005 at the University of Franche-Comté. He is currently an HPC research engineer at the computing centre of Université de de Franche-Comté. His research interests include parallel computing, distributed systems and numerical asynchronous algorithms.

Nicolas Marilleau obtained his Ph.D degree in 2005 at the University of Franche-Comté. He is Research Engineer at the UMMISCO International Research Unit of the IRD (Research Institut for Development). His main research area is modeling and simulating complex system with multi-agent systems and distributed computing. He applies his work especially to soil sciences and geography, domains where he is actively involved to design and develop models representing real phenomena.