A Communication Schema for Parallel and Distributed Multi-agent Systems Based on MPI

Alban Rousset(✉), Bénédicte Herrmann, Christophe Lang, and Laurent Philippe

Femto-ST Institute, University of Franche-Comté/CNRS, Besançon, France
{alban.rousset,benedicte.herrmann,christophe.lang, laurent.philippe}@femto-st.fr

Abstract. The interest for Multi-Agents Systems (MAS) grows rapidly and especially in order to simulate and observe large and complex systems. Centralized machines do not however offer enough capacity to simulate the large models and parallel clusters can overcome these limits. Nevertheless, the use of parallel clusters implies constraints such as mono-threaded process of execution, reproducibility or coherency. In this paper, our contribution is a MPI based communication schema for Parallel and Distributed MAS (PDMAS) that fits High Performance Computing (HPC) on cluster requirements. Our communication schema thus integrates agent migration between processes and it guarantees message delivery in case of agent migration.

Keywords: Multi-Agent simulation · Parallelism · Communication schema

1 Introduction

The interest for Multi-Agents Systems (MAS) grows rapidly and especially in order to simulate and observe large and complex systems. Centralized machines, like desktop computer, do not however offer enough capacity to simulate the model expected: their lack of memory or their processor is not powerful enough. Parallel machines like clusters or networks of workstations can overcome these limits. Nevertheless, using a cluster or a network of workstations implies management of some constraints, such as distribution, load balancing, migration, coherency or inter-processors communications, that we do not have on a single workstation. Efficiently using these platforms, to get good performance, also requires the relay of adapted software stacks. This clearly means that the MPI interface [9] must be used as a base for parallelism and communication.

Using MPI with its mono-threaded process execution model to run MAS is however a challenge. The contribution of this paper is a MPI based communication schema for Parallel and Distributed MAS (PDMAS) that fits High Performance Computing (HPC) on cluster requirements. Our communication schema

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 442–453, 2015.
DOI: 10.1007/978-3-319-27308-2_36

thus integrates agent migration between processes and it guarantees message delivery in case of agent migration.

This article is organized as follows. In Sect. 2, we detail the related work on Multi-Agent Systems and parallel execution context. Then, in Sect. 3, we identify some lacks or limits in existing Parallel and Distributed Multi-Agent platforms. In Sect. 4, we give an overview of our proposition and we detail it in Sect. 5 for the communication schema and Sect. 6 for the proxy system used to follow mobile agents. We present the performance results obtained with our proposition, based on the same model used to make the survey on PDMAS [11], in Sect. 7. We finish the paper with conclusions and future work.

2 Related Work

Multi-Agent Systems are platforms which provide support to run simulations based on several autonomous agents [7]. Among the most known platforms we can cite: NetLogo [13], MadKit [8], Mason [10] and Gama [12]. For large models, these platforms are sometimes no longer sufficient to run simulations in terms of memory and computation power. This is, for example, the case in simulating individual behaviour of urban mobility [3] in a large city. In some cases increasing the size or the precision of models is however necessary to find emergent behaviours that we would never expect or never have seen otherwise. For this reason several Parallel and Distributed Multi-Agent Platforms exist such as RepastHPC [5], D-Mason [6], Pandora [1] and Flame [4] or JADE [2]. These platforms provide a native support for parallel execution of models. That is to say, support for collaboration between executions on several physical nodes, distribution of agents, communications between agents and so on. All existing PDMAS platforms propose mechanisms to communicate between agents during the simulation. But, this is done only with agents which are executed on the same process or in the buffer zone, the zone shared between two adjacent processes. For example, RepastHPC proposes mechanisms to request a copy of a remote agent from other processes, but if the copy agent is modified, modifications are never reported in the remote agent. In other words there are no synchronization mechanisms to apply modifications nor communication mechanisms to communicate directly with remote agents. Only the Flame [4] platform allows to communicate with remote agents executed on other processes. To perform the inter-process communications, Flame platform uses its own communication library based on MPI which is called Message Board Library called *libmboard*. Each process which participates in the simulation has an instance of *libmboard* in order to make synchronizations and to perform communications. One of the advantages of the *libmboard* is that sending and receiving messages is a non-blocking process. It then allows much of the communication time to be overlapped with computations. The Flame platform however offers low performances on clusters compared to others platforms (RepastHPC, D-MASON...) as shown in [11]. In addition Flame uses a proprietary programming paradigm (X-Machine) that could not be easily adopted by modellers used to standard languages as C or Java.

3 Implementing PDMAS on HPC Platforms

Targeting high performance computing implies some constraints on MAS implementation. Usually, PDMAS platforms are implemented using a Single Process Multiple Data (SPMD) programming paradigm in order to provide scalability. MAS simulations generally involve several tens of thousands of agents which potentially communicate with each other at each simulation time step. The communication bottleneck is thus a key problem as, in a parallel context, the running time is affected by the frequent communications. As the de facto standard of communication infrastructure in HPC cluster is the message passing interface (MPI) it is important to take care of the communication primitive properties in order to reduce the communication overhead. This constraint combines with another which is to have only a single mono-threaded process of execution on each allocated core. This constraint is imposed by most batch systems as SGE or SLURM. Using a single process of execution implies that we cannot use mechanisms like *listener*, *onEvent* or *onMessage* to communicate because we cannot dedicate one thread to wait for a message during the agent set execution. Messages must thus either be received by issuing non-blocking receives during the execution or at the end of the execution with a blocking receive. This illustrates the complexity of using a mono-threaded execution model to implement asynchronous communications.

Another problem set by the parallel context is the problem of message delivery in a time-driven MAS. In time-driven simulations the simulation is divided in time steps that represent the temporal discretization of the simulation. Messages between agents are thus bounded by time step. The issue is: at what time step must the message be delivered? To guarantee reproducibility, each message sent at time step n must be received before the beginning of time step $n + 1$. Indeed, as processes run asynchronously from each other, delivering messages in the same time step n cannot be guaranteed: the receiving agent can either be scheduled later or sooner in different runs, depending on the node load. For this reason delivering messages at the beginning of time step $n + 1$ is the only way to guarantee that a message will always be received at the same time step.

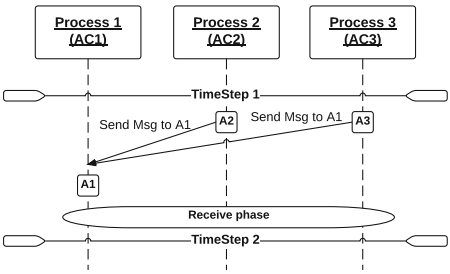

Fig. 1. Indeterminism of message order

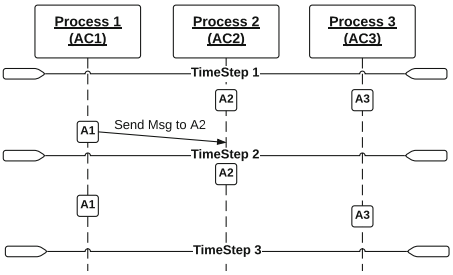

Fig. 2. Indeterminism in time step

Note however that, even with delivering messages at time step $n + 1$, the stochastic nature of MAS makes it difficult to provide an absolute guarantee during the simulation as illustrated on Figs. 1 and 2. Figure 1 shows a case of indeterminism of message receive order. Let p_1, p_2 and p_3 be three processes executing a part of a simulation. If both p_2 and p_3 send a message at the same time to p_1, we cannot know in which order we need to apply them on p_1. On a centralized system these messages could be stamped with a clock value that differentiate them. In a parallel or distributed context we cannot relay on this clock value. Figure 2 illustrates the need for receive phases between time steps. Agent $A1$ is scheduled at the end of the time step and sends a message to agent $A2$ at that time. If the message is delayed on the network and agent $A2$ is scheduled at the beginning of the next time step. Then agent $A2$ may miss receiving its message at time step $n+1$. For this reason, it is important to define receive phases at the end of each time step.

As underlined previously, the full functionality of being able to communicate with every process of the simulation, is only supported by Flame while other PDMAS limit it to the local process or neighbour processes. We advocate for inter-agent remote communications in PDMAS for two reasons. First, in models focusing on individual motions as in a city/urban mobility, agents may need to keep communicating with their contacts while moving. Due to the distribution of the simulation, agents could move anywhere on the environment, on different processes, and thus must be able to communicate with every process. Second, on graph based models, limiting agent communication to the neighbourhood leads to complex mapping constraints: non-planar graphs cannot easily be mapped on grids while keeping neighbourhood constraints.

From these reflections, we propose a communication schema for PDMAS in order to allow local and distant communications between agents without paying attention to agent location. Our communication schema allows reproducibility and guarantees that each message sent in time step n is received at the beginning of time step $n + 1$.

4 Proposition

Before presenting the communication algorithm, we give information about the Parallel and Distributed Platform that we develop. In this platform, the simulation is divided in n parts and each of these parts are distributed on p processes to perform the simulation in parallel. These processes are called *Agent Containers* (AC) and are in charge of scheduling and executing agents, of receiving messages from agents executed on other processes and of delivering these messages to its own agents. ACs also manage movement of agents between processes.

In Multi-Agent Simulations, agents must be identifiable. In PDMAS agents must be identifiable regardless of their process. For this reason we associate a *System ID*, inspired from RepastHPC and presented in Fig. 3, to each agent. This *System ID* is composed of four values: a global unique ID (GID), the ID of the process which created the agent (OwnProc), the ID of the process on which

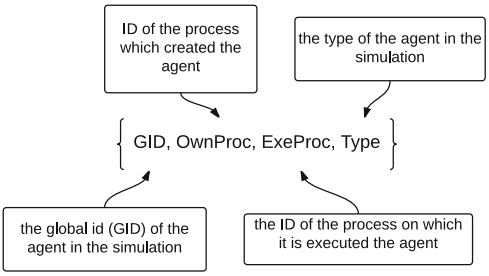

Fig. 3. Schema of the identification of an agent

the agent is currently executing (ExeProc) and the ID of the agent type (Type). With this System ID, we can know at any time where an agent has been created and where the agent is currently executed.

ACs execute four different phases at each time step to implement all the communication requirements: (1) Run agent's behaviours, (2) Receive messages, (3) Migrate agents, and (4) Update agents. In this paper we only focus on phases 2 and 4, that is to say sending and receiving messages and agent updates to perform communications even if they move on the environment. Phase 1 does not differs from other MAS and phase 3 is not necessary for understanding. The communication schema, the core of the contribution, is presented in the two next sections. Section 5 details the communication schema between agents and Sect. 6 details remote communication with mobile agents.

5 Communication Schema (Receive Message Phase)

As said previously, for coherency and reproducibility reasons, every message sent at time step n must to be received at the beginning of time step $n + 1$. Due to the stochastic nature of agents, we cannot however know how many messages must received, and from how many processes, at the end of a time step and so we cannot know when processes can proceed the next time step. For this reason we must use a termination algorithm. MPI proposes synchronization mechanisms like *barrier*, which are synchronization points, but they do not solve this problem. If we use a *barrier* to bound time steps, some faster processes will reach the synchronization point and then block until the last process reach it. Processes that did not reach the barrier could however send new messages that will not be processed by the recipient processes as they are blocked on the barrier. Thus messages are lost. This is the reason why mechanisms propose by MPI are no longer sufficient in this case.

To overcome this problem, we use a termination algorithm to reach an agreement between processes to switch to the next time step at the right time: when all processes have terminated to process the current time step. Our termination algorithm is based on a bi-directional ring with a coordinator. In our case, we use the bi-directional ring to check that all processes have terminated their messages

receipt and that all processes can proceed to the next time step. We decided to use a bi-directional ring instead of a single ring, for efficiency because we divided the path in two executed in parallel.

In our proposition, agents send messages in their behaviours using method like *communicate(MSG)* but they do not directly receive messages. As said before ACs are responsible of receiving messages and to deliver these messages to agents. To proceed to the next time step, we need to be sure that each sent message is received. For this reason, an AC needs to acknowledge (*Ack*) each message of each agent in order to be sure that they are no pending messages (Fig. 4).

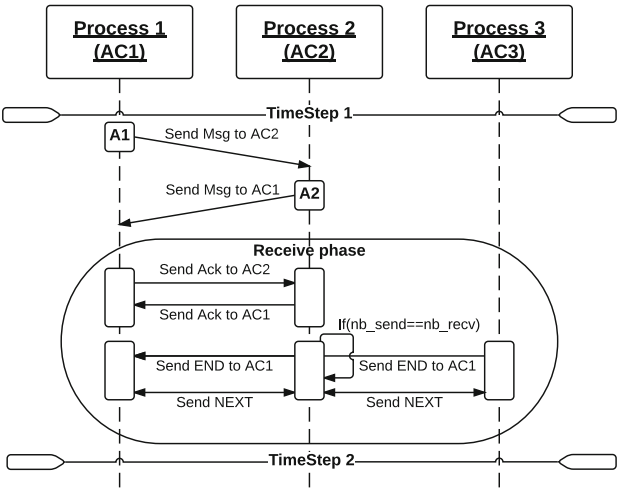

Fig. 4. An instance of receiving phase executed at each time step by each container of agent

The AC algorithm consists in waiting the *Ack* for all messages sent by its agents. Then it sends an END message to a coordinator, to inform that the AC has terminated the processing of its messages and wait to receive messages from the coordinator or from others processes. When the coordinator has received an END message from all the processes of the simulation, he sends a $NEXT$ message using the bi-directional ring termination algorithm. At the end of the termination algorithm, all processes are sure that they can execute the next time step. The complete algorithm is presented in Algorithm 1.

By this way time steps are bounded by received messages phase. This algorithm works fine for agents without mobility. In case of agent mobility an AC needs to know where the target agent is run to deliver a message as agents may moves from a process to another. We explain the algorithm used to overcome this problem in the next section.

```
if (AC is coordinator) then
    while ((NbSendMessage ≠ NbAckReceived) && (nbEndReceived ≠ NbProcesses-1))
    do
        RecvMsg(Msg)
        switch Msg.TAG do
            case DATA_MSG  DeliverApplyMSG(Msg) ; SendAck(Msg.Source);
            case ACK NbAckReceived++;
            case END nbEndReceived++;
        endsw
    end
    SendNextTimestep(⌈NbProcesses/2⌉);
    WaitForTermination();
else
    while ((NbSendMessage ≠ NbAckReceived) && (NextTimestep ≠ false)) do
        RecvMsg(Msg);
        switch Msg.TAG do
            case DATA_MSG DeliverApplyMSG(Msg);SendAck(Msg.Source);
            case ACK NbAckReceived++;
            case NEXT NexTimestep ← true;
        endsw
    end
    if (AC.ID = ⌈NbProcesses/2⌉) then
        SendNextTimestep(PREVIOUS_AC);
        SendNextTimestep(NEXT_AC);
    else if (AC.ID > ⌈NbProcesses/2⌉) then
        SendNextTimestep(NEXT_AC);
    else
        SendNextTimestep(PREVIOUS_AC);
    end
end
```

Algorithm 1. Receiving phase executed at each time step by each AC

6 Proxy System (Agents Update Phase)

Multi Agent Systems often use mobile agents: agent that are not fixed on the environment but rather move. This is, for instance, the case of wolves and sheep in the classical prey-predator model. In PDMAS it is necessary to distribute the environment on several processes. With agent mobility agents may move from a process (or AC) to another (process or AC) to keep the continuity of the environment and thus perform its behaviour. If we want to send a message to an agent, we need to know on which process the agent is now run. To respect the single threaded process constraint of HPC context, we use a Proxy System (PS) which consists in letting a trace of each agent on the process which creates it at the beginning of the simulation. This trace is updated during the simulation.

Algorithm 2 details how the proxy for agents is updated. We update PS when an agent move from a process (or AC) to another (process or AC). Each AC contains an hashmap of proxies for each agent that it creates at the beginning of the simulation. When an agent is going to move, the container looks if this agent have been created by itself thanks to the *System ID* of agents. If it is an agent that the process (or AC) has created, the AC changes, in its proxy hashmap, the current process of the agent where the agent is executed to the future process on which the agent will be moved. On the other cases, the AC sends a message (that contains the future process on which the agent will be moved) to the creator of

this agent in order to inform the AC that one of its agent move from a process (or AC) to another (process or AC).

This phase of proxy update depends on the same termination rules as the message receipt, because we cannot know how many updates we receive and from how many processes we can receive updates. For this reason we also use the previously termination algorithm base on bi-directional ring. As this update phase is completed before we start a new time step then the proxy hashmap are always up-to-date during the agent run step.

```
while ((!End of Migration) && (nbEndReceived ≠ NbProcesses-1)) do
    if (Agent.GetOwnerProc() = AC_ID) then
        MapProxy[Agent.GID] ← AC_ID;
    else
        SendMsgMajProxy(Agent.GetOwnerProc(), Agent.FutProcess);
        NbSendMigration++;
    end
end
if (AC is coordinator) then
    while ((NbSendUpdates ≠ NbAckReceived) && (nbEnd ≠ NbProcesses-1)) do
        RecvMsg(Msg);
        switch Msg.TAG do
            case DATA_UPDATE_AGENT MapProxy[Msg.GID] ← Msg.FutProcess;
            SendAck(Msg.Source);
            case ACK_UPDATE NbAckReceived++;
            case END nbEndReceived++;
        endsw
    end
    SendFinishUpdating(⌈NbProcesses/2⌉);
    WaitForTermination();
else
    while ((NbSendUpdates ≠ NbAckReceived) && (FinishUpdating ≠ false)) do
        RecvMsg(Msg);
        switch Msg.TAG do
            case DATA_UPDATE_AGENT MapProxy[Msg.GID] ← Msg.FutProcess;
            SendAck(Msg.Source);
            case ACK_UPDATE NbAckReceived++;
            case NEXT FinishUpdating ← true;
        endsw
    end
    if (AC.ID = ⌈NbProcesses/2⌉) then
        SendFinishUpdating(PREVIOUS_AC);
        SendFinishUpdating(NEXT_AC);
    else if (AC.ID > ⌈NbProcesses/2⌉) then
        SendFinishUpdating(NEXT_AC);
    else
        SendFinishUpdating(PREVIOUS_AC);
    end
end
```

Algorithm 2. Updating phase executed at each time step by each AC

In this way, we always know where an agent is executed by interrogating and sending messages to the process that creates the agent at the beginning of the simulation. If the agent is not executed on this process then the process which receives the message forwards this message to the right process on which the agent is now run (Fig. 5). The message will only be acknowledged by the right container process. As a process waits until its gets all its acknowledgements

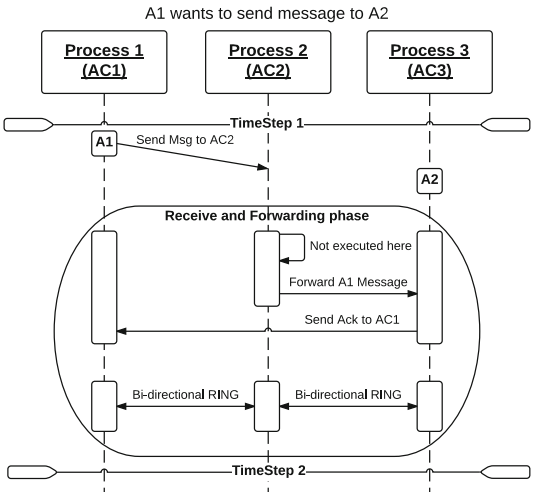

Fig. 5. An instance of updating phase executed at each time step by each AC

before sending its END message, then it guarantees that the sent message will be delivered at time step $n + 1$. In addition, for the user, this is an hidden functionality, he does not pay attention to where is the agent located, local or remote. The only mandatory information is the *System ID* of the agent you want to contact and the message will be received.

7 Experimentation

In this section we present some results using our communication schema for Parallel and Distributed Multi-Agent simulation. We have implemented the communication schema in the Parallel and Distributed Multi-Agent Platform called FractalPMAS that we develop. To assess the performance of the communication schema, we have implemented a reference model defined and already implemented in most known PDMAS platforms in [11]. This model respects the main properties that can be found in Multi-Agent Systems: perception, communication, mobility. In this model each agent is composed of 3 behaviours, performed at each time step: walk behaviour which allows agents to move in a random direction on the environment, interact behaviour which allows agents to interact and send messages to other agents and finally compute behaviour which allows agents to generate a workload.

In this reference model we also have implemented a way to evaluate the performance of our remote communication schema between agents. That is to say, instead of sending message to agents in the perception field, each agent sends messages to randomly chosen agents which are run on an other process if they are in its perception field.

About the HPC experimental settings: we have run the reference model on a 764 core cluster. Each node of the cluster is a bi-processor, with Xeon E5 (8*2 cores) processors running at 2.6 Ghz frequency and with 32 Go of memory. The nodes are connected through a non blocking DDR InfinyBand network organized in a fat tree.

Execution results on the scalability of a 10 000 agents model are given on Fig. 6, with the ideal speed-up reference. Note that the reference time used to compute the speed-up is based on a 2 cores run of the simulations as RepastHPC cannot be run on a single core. To assess scalability we vary the number of cores used to run the simulations while we fix the number of agents. We then compute the obtained speed-up. Figure 6 represents the scalability of the platform with our communication schema for local and remote communications between agents.

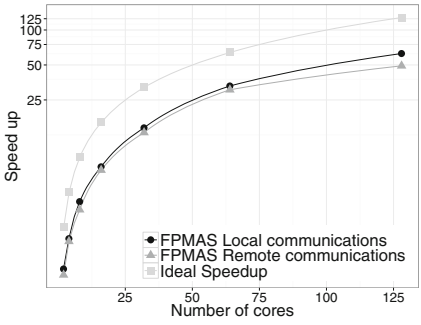

Fig. 6. Scalability of FPMAS platform running simulations using 10 000 agents, FFT 100 and 200 cycles

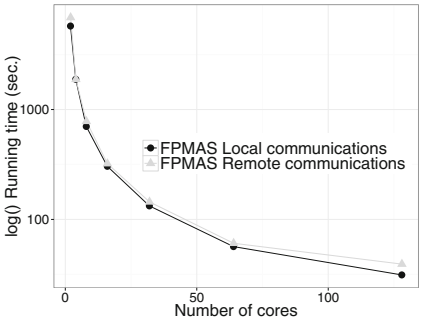

Fig. 7. Running time for FPMAS platform running simulations using 10 000 agents, FFT 100 and 200 cycles

As we can see both scale well, even if the local communications scales better. The difference between two speedup are not clearly noticeable. Obviously, remote communication offers lower performance due to the intensive exchanges between processes. Figure 7 represents the running time of simulations for 200 time steps and for local and remote communications.

Figure 8 represents the workload of our proposition on 8 cores and we vary the number of agents from 1000 to 30 000, and comparing the local communication running time and the remote communication running time.

Obviously, remote communications cannot be better than local communications, but remote communications support well the workload. For example, for 10 000 agents there is a difference of $\tilde{1}0\,\%$ between running time for local and remote communications and for 30 000 agents there is a difference of $\tilde{3}0\,\%$. These values are acceptable but we need to improve mechanisms. This implementation is only a proof of concept.

To assess the viability and the performance of our communication schema, we compare in Table 1 the running time for the model defined previously [11]

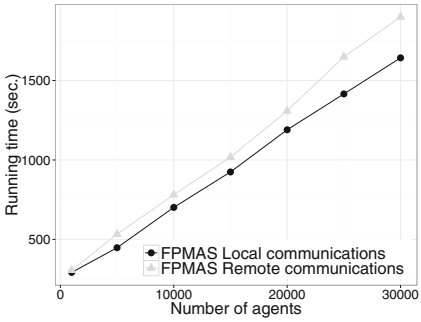

Fig. 8. Workload behaviour for FPMAS platform running simulations using 8 cores

Table 1. Comparison of running time for platforms studied in [11] with 10,000 agents

Cores	4	8	16	32	64	128
FPMAS (local com.)	2097.09	877.59	380.09	175.96	66.63	26.32
FPMAS (remote com.)	2142.68	893.6025	391.52	185.8075	67.0575	39.2125
RepastHPC	8772.93	3276.91	1277.03	724.99	497.53	367.76
Flame	17418.79	9282.20	4773.86	2428.22	1520.65	836.29

with performance obtained for other PDMAS platforms. In terms of running time FPMAS is better than other platforms. Communications (local or remote) does not impact clearly the results on this model. Compared to others platforms, better results could be explain by the way used to modelled and to structure the agents and the simulation compared to other platforms.

8 Conclusion and Perspectives

In this paper we have presented a communication schema for Parallel and Distributed Multi-Agent simulation that fit the constraints set by HPC systems. This communication schema is based on the MPI communication interface and allows communication with local and remote agents. Our contribution aims at proposing a communication schema which offers more efficiency while guaranteeing properties as reproducibility and coherency.

In our future work, we intend to better examine the efficiency of synchronization using our communication schema and also to improve the scalability of the communication schema. More improvements could be made in the implementation which is only a proof of concept. Then we will use this platform to assess load balancing in PDMAS.

Acknowledgement. Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté.

References

1. Angelotti, E.S., Scalabrin, E.E., Ávila, B.C.: Pandora: a multi-agent system using paraconsistent logic. In: Computational Intelligence and Multimedia Applications, ICCIMA 2001, pp. 352–356. IEEE (2001)
2. Bellifemine, F., Poggi, A., Rimassa, G.: Jade-a fipa-compliant agent framework. In: Proceedings of PAAM, vol. 99, p. 33, London (1999)
3. Chipeaux, S., Bouquet, F., Lang, C., Marilleau, N.: Modelling of complex systems with aml as realized in miro project. In: 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), vol. 3, pp. 159–162 (2011)
4. Coakley, S., Gheorghe, M., Holcombe, M., Chin, S., Worth, D., Greenough, C.: Exploitation of hpc in the flame agent-based simulation framework. In: Proceedings of the 2012 IEEE 14th International Conference on HPC and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, HPCC 2012, pp. 538–545. IEEE Computer Society, Washington, DC (2012)
5. Collier, N., North, M.: Repast HPC: A Platform for Large-Scale Agentbased Modeling. Wiley, Hoboken (2011)
6. Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., Spagnuolo, C.: A framework for distributing agent-based simulations. In: Alexander, M., et al. (eds.) Euro-Par 2011, Part I. LNCS, vol. 7155, pp. 460–470. Springer, Heidelberg (2012)
7. Ferber, J., Gutknecht, O., Michel, F.: From agents to organizations: an organizational view of multi-agent systems. In: Giorgini, P., Müller, J.P., Odell, J.J. (eds.) AOSE 2003. LNCS, vol. 2935, pp. 214–230. Springer, Heidelberg (2004)
8. Gutknecht, O., Ferber, J.: Madkit: a generic multi-agent platform. In: Proceedings of the fourth international Conference on Autonomous agents, pp. 78–79. ACM (2000)
9. Hempel, R., Hey, A.J., McBryan, O., Walker, D.W., et al.: Message passing interfaces. Parallel Comput. **20**(4), 415–416 (1994)
10. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K.: MASON: a new multi-agent simulation toolkit. Simulation **81**(7), 517–527 (2005)
11. Rousset, A., Herrmann, B., Lang, C., Philippe, L.: A survey on parallel and distributed multi-agent systems. In: Lopes, L., et al. (eds.) Euro-Par 2014, Part I. LNCS, vol. 8805, pp. 371–382. Springer, Heidelberg (2014)
12. Taillandier, P., Vo, D.-A., Amouroux, E., Drogoul, A.: GAMA: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In: Desai, N., Liu, A., Winikoff, M. (eds.) PRIMA 2010. LNCS, vol. 7057, pp. 242–258. Springer, Heidelberg (2012)
13. Tisue, S., Wilensky, U.: Netlogo: design and implementation of a multi-agent modeling environment. Proc. Agent. **2004**, 7–9 (2004)