Agent Migration in HPC Systems Using FLAME*

Claudio Márquez, Eduardo César, and Joan Sorribes

Computer Architecture and Operating Systems Department (CAOS),
Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain
claudio.marquez@caos.uab.es, {eduardo.cesar,joan.sorribes}@uab.cat
http://caos.uab.es/

Abstract. In HPC agent based applications, the complex interaction rules would likely cause workload imbalances that negatively affect the simulation time. In addition, the imbalance problem may vary during the application execution in accordance to the behavior of the agents. Consequently, a solution to this problem should be able to dynamically balance the load. A dynamic load balancing scheme could be based on migrating agents between processing units. In this paper, we propose a modification of the agent-based simulation framework FLAME that provides the automatic generation of the routines needed to dynamically migrate agents among different computational units. However, most agent-based simulation frameworks do not include routines for migrating agents. Moreover, we demonstrate their use in a simple load balancing scheme on a specific application.

Keywords: Agent-based Simulation, FLAME, Migration Agents, SPMD, Load Balancing, Application Tuning.

1 Introduction

Agent-Based Modelling (ABM) is a popular technique for simulating complex systems in several domains such as economical, medical, biological and social science. Basically, Agent-Based Modelling and Simulation (ABMS) describe the system's behavior through the interactions of a set of autonomous entities in the environment.

Large ABMS applications facilitate the study of complex problems. In this case, they should be deployed in a High Performance Computing (HPC) environment to provide sufficient resources and be executed in a reasonable amount of time because in most cases ABMS are CPU intensive and require large amounts of memory [1].

Moreover, agent based applications show significant variations in the amount of computing and communication time. These variations are given by the large amount of interactions among agents, and the different rules of behavior exhibited by most of these models. Moreover, the evolution of the simulation may

* This work was funded by MICINN Spain, under the project TIN2011-28689-C02-01.

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 523–532, 2014.

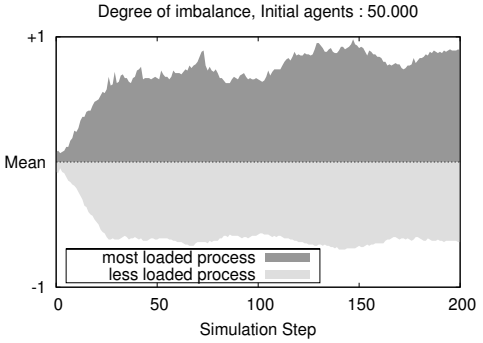


Fig. 1. Degree of computing imbalance during the simulation of the SIR model

produce dynamic changes in the workload. Therefore, during the execution of the simulation, load imbalances are likely to appear. Figure 1 shows the degree of imbalance of a simulation of 50.000 agents in a cluster of 128 processes.

In order to solve such problems, schemas for dynamically adjusting the load should be used. However, it would be very helpful for such dynamic schemas that the parallel simulation environments include mechanisms that allow the migration of agents between different computational units.

Currently, a few ABMS parallel applications oriented to HPC environments can be found. Ecolab [2] is an object-oriented environment written in C++ and MPI. Repast HPC [3] was recently released in 2012, and written in C++ using MPI for parallel operations. Contrary to Ecolab, Repast HPC was created from the beginning for large-scale distributed computing platforms. Although both Ecolab and Repast HPC are based on a Single Program Multiple Data (SPMD) paradigm, they do not include generic migration routines, so the developer should deploy the whole migration code. Moreover, D-Mason [4] is a framework written in Java, based on a master/worker paradigm. D-Mason is focused on using the idle desktop workstations and subdividing the workload among these heterogeneous machines. In addition, recent improvements [5] of D-Mason provide load balancing schema. Finally, FLAME [6] framework allows the production of automatic parallelizable code to run on a large HPC infrastructure. This framework is based on a SPMD paradigm, it has been continuously developed since 2006, and it used for economic modelling in the EURACE [7] project. FLAME is written in C using MPI, and is aimed principally at the economical, medical, biological and social science domains.

Current implementations of dynamic load balancing strategies are often developed using centralized or hierarchical approaches. On one hand, centralized approaches commonly report a high computational cost and scalability problems. In the other hand, decentralized approaches can present problems regarding the quality of the balance because the neighboring processes exchange incomplete information. In [8] a centralized load balancing based on space repartitioning is proposed. In [9] a hierarchical multi-level load balancing strategy is presented,

and centralized and hierarchical schemas are compared. In [10] a decentralized algorithm based on moving the boundary between the neighbor region is defined. In [11] three algorithms using recursive domain decomposition in a binary tree structure are compared using balance speed and communication costs. In [12] a complex partitioning approach based on irregular spatial decompositions is presented. In [13] distributed cluster-based partitioning and load balancing schema for problems of flocking behaviors are defined. In its current version, the code generated by FLAME lacks the necessary routines to allow the migration of agents; therefore, all migration routines should be implemented directly on the generated code.

This paper describes a modification of the FLAME framework for automatically generating, from the agents's specification, the routines necessary for migrating agents between different computational units. The approach aims to implement dynamic policies for deterministic agent-based simulations, which define the agents as state machines.

The rest of this document is organized into five sections. First, Section 2 briefly describes FLAME. Next, the migration routines are presented (3). The results section shows how these routines have been used for implementing a simple load balancing schema for a SIR ABMS application (4). The final section includes the conclusions and the future work(5).

2 FLAME

In contrast to common ABMS environments, FLAME is a tool that enables the necessary source code to be generated for the simulation; hence, it is not a simulator in itself. FLAME automatically generates the simulation code in C through a template engine. In the same way, FLAME provides a set of template files that the template engine uses to generate the simulation code getting information from the model definition. The model definition consists of X-Machine Markup Language (XMML) files, which is a dialect of XML and the implementation of the agent functions contained in C files. The migration routines are automatically generated from a set of extra template files as shown in figure 2(b). Figure 2(a) shows the files required by the FLAME framework to create the simulation code, and figure 2(b) shows the FLAME diagram including the proposed improvement.

2.1 Functional Description

The deployment of agents is based on finite state machines called X-machines, which consist of a finite set of states, transitions between the states and actions. To perform the simulation, FLAME keeps each agent as an X-machine data structure, whose state is changed via a set of transition functions. Furthermore, the transition functions perform message exchanges between agents if necessary. Then, the simulation environment is composed mainly by a set of X-machines defined by their state transitions, internal memory, and agent messages.

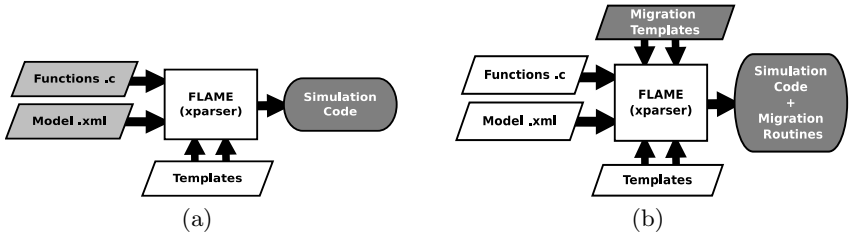


Fig. 2. Base-diagram of the FLAME framework and diagram with the improvement

The X-machines are kept in linked lists, one for each state of a specific agent. When the simulation starts, the X-machines are inserted into the list related to the initial state. Then the corresponding transition function is applied to each X-machine. Once this has been done, these X-machines are inserted into the list related to the next state. This process is repeated until the last state, which determines the end of the iteration is reached.

2.2 Parallelization

In HPC environments, FLAME communications are managed by the Message Board Library *libmboard*, which uses *MPI* to communicate between processes. *Libmboard* handles the messages of the agents through message managing mechanisms and filtering before sending messages to local agents and agents belonging to external processes. FLAME handles deadlocks through synchronization points, which ensure that all the data is coordinated among agents using an SPMD pattern.

Figure 3(a) shows the communication pattern between local agents and external agents using *libmboard* library. This library sends all messages to the agents through a coordinated communication between different *MPI* processes.

The parallel distribution of the agents in FLAME is based on two static partitioning methods: geometric partitioning and round-robin partitioning. Currently, FLAME does not have mechanisms to enable the movement of agents between processes. Thus, the workload in each parallel process will rely on the initial population of agents.

Consequently, evolution of the simulation may produce computation imbalance causing overhead, and also may produce excessive external communications due to the interaction among agents (as shown in figure 3(b)). Therefore, the time required to complete the simulation will be negatively affected.

3 Migration of Agents

An agent migration mechanism is necessary to implement policies in order to solve load/communication imbalance problems. However, developing this mechanism can be a time consuming task because, among other reasons, the programmer must understand the code generated by the framework for each specific

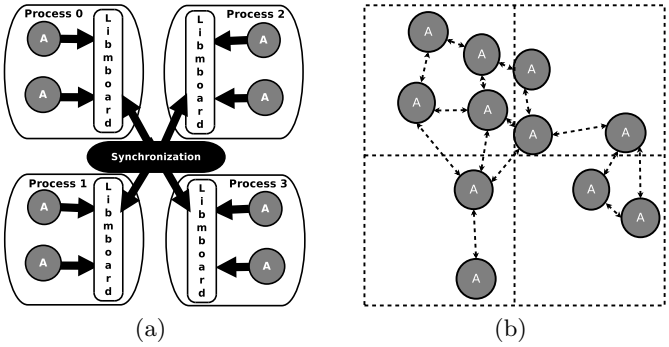


Fig. 3. *(a)* Parallel communication and synchronization via libmboard. *(b)* Workload problems associated with the distribution of agents.

simulation code. Consequently, enhancing the tool for automatically generating efficient routines for migration agents could be very helpful.

In order to achieve this new feature, the developer should only add three new templates for generating the migration routines. Then, the template engine processes these templates to obtain the information about the agent model from the definition file. This information includes issues such as: agent types, the variables (properties) of the agents and the size of each agent's variable.

Internally, the template engine interprets the template's information using the information about the agent model, and generates migration routines with the simulation code (as shown in figure 2(b)). Once the simulation code has been created, the migration routines can be used for automatically adding and removing agents from any process. The migration process can be subdivided into two procedures: *contribution* and *acquisition*.

On one hand, **contribution procedure** consist of removing, packing and sending the selected agents in the sender processes. In this procedure, the agents to be migrated are held in a set of *linked lists* identified by the *id* of the target process. Later, the agents are stored in a set of contiguous memory buffers to be packed. At this point, the buffer sizes have been automatically calculated and created by the routines. Finally, the sender processes will have a collection of buffers to be sent as agent packages (one for each recipient process).

On the other hand, the **acquisition procedure** consists of receiving, unpacking and adding the agents in the recipient processes. The packages of agents are stored in buffers to be unpacked. Once the agents have been unpacked, they are inserted with the other agents in the recipient process. Algorithms 1 and 2 show the procedures involved during the migration of agents.

In order to send the agents to a recipient process in a single communication, the agents need to be stored in contiguous memory before being sent. This migration is accomplished by packing and unpacking data using *MPI* functions. Before being used, these *MPI* functions require memory buffers whose sizes depend on the type and amount of agents. Additionally, the sending and receiving

Algorithm 1 Contribution procedure

```
1: while agents to be sent do
2:     remove of the current process
3:     insert in the recipient list
4: calculate buffer sizes
5: pack the agents
6: send the agents
```

Algorithm 2 Acquisition procedure

```
1: ask how many agents should receive
2: preparation of memory buffers
3: receive the agents
4: while packed agents do
5:     unpack agent
6:     insert in the current process
```

of the agents has been deployed using *MPI* asynchronous functions to overlap the costs of communication and computation.

Before performing the migration process, a criterion must be established to decide which agents should be sent. Then the migration process starts through the migration routines mentioned in section 3.1. The migration process should also be required to decide when it should be performed. Nevertheless, this partially depends on the criterion by which the agents were selected. Below, the main migration features incorporated into FLAME are described.

3.1 Migration Routines

The migration routines are specifically generated for each type of agent in the model, so it is possible to perform migrations after any transition. The following list introduces the main migration routines. The prefix NAME indicates the name of a specific type of agent.

- *Init_migration*: Initializes global variables and data structures involved in the migration.
- *Pop_NAME*: Moves agents to a specific *linked list* and removes them from the current process.
- *Pack_NAME*: Packs all agents kept in the *linked lists* in contiguous memory buffers, one for each recipient.
- *send_NAME/recv_NAME*: These routines are prototypes to define how to send and receive agents.
- *Unpack_NAME*: Unpacks the packed agents received. Then, inserts the received agents to the X-machine list of the current process.
- *Push_NAME*: Add an agent to the current process.

With these functions inserted into FLAME, it is possible to incorporate a load balancing schema based on the migration of agents between processes.

4 Experimental Results

The main objective of this section is to demonstrate that, using the migration routines, it is possible to migrate agents and correct imbalance problems in ABMS applications. For testing purposes only, the SIR epidemic model described

below is used [14]. The simulation workload changes dynamically due to the random distribution of death and birth rates. In this experiment, the initial distribution of agents is based on a round-robin distribution and performed during 200 simulation steps.

The SIR model describes the spread of an epidemic within a population on a 2D toroidal space. The population is divided into three groups: the Susceptible (S), the Infectious (I), and the Recovered (R). For this reason, this model is called SIR. In summary, the susceptible are those individuals who are not infected and not immune, the infectious are those who are infected and can transmit the disease, and the recovered are those who have been infected and are immune. Additionally, natural births and deaths during the epidemic are included in this SIR model, so individuals could die from the disease or by natural death due to aging. Consequently, births and deaths represent a dynamic creation and elimination of agents; therefore, the workload can change as the simulation proceeds.

In this example, generated functions are tagged as `Person` because agents are defined as `Person` in the model specification. First, the function call *Init_migration()* is inserted before the configuration section of the *libmboard* access mode. Then, all configuration sections of the *libmboard* access mode are set up in *MB_MODE_READWRITE* mode.

Algorithm 3. Overview of the load balancing schema described in [15]

1: collect all computing times for each process
2: $tolerance \leftarrow threshold * avg\_time$
3: **if** $\forall i \in procs, \exists\ proc_i / |imbalance(proc_i)| \geqslant tolerance$ **then**
4: sort computing times in descending order
5: $i, j \leftarrow$ index of the first and last process in the *sorted list*, respectively
6: **while** $|imbalance(proc_i \wedge proc_j)| \geqslant tolerance$ **do**
7: $contribution\_range_i \leftarrow exceeded\_time_i \pm tolerance$
8: $j \leftarrow$ index of the last process in the *sorted list*
9: **while** $|imbalance(proc_i \wedge proc_j)| \geqslant tolerance$ **do**
10: $acquisition\_range_j \leftarrow required\_time_j \pm tolerance$
11: calculate expected migration for $proc_i | proc_j$
12: sort underloaded computing times
13: **if** $|imbalance(proc_i)| \leqslant tolerance$ **then** break
14: $j - -$
15: sort overloaded computing times
16: $i \leftarrow$ index of the first process in the *sorted list*
17: **if** $|imbalance(proc_i)| \leqslant tolerance$ **then** break
18: Execute the asynchronous exchanges

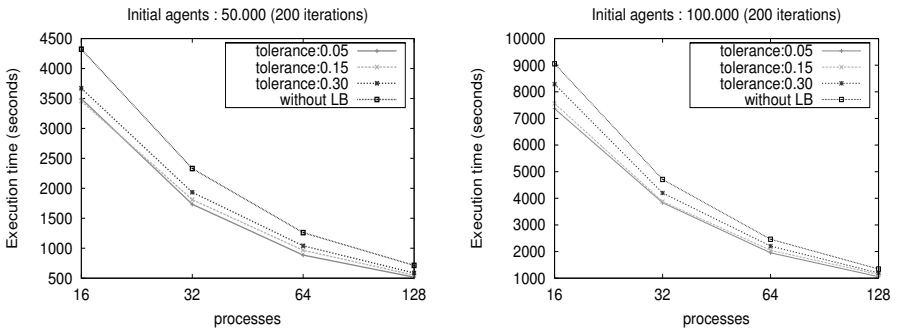
For the **load balancing schema**, we used a schema for SPMD applications described in [15]. This schema dynamically decides the global reconfiguration of the workload using an imbalance threshold, a computing time, and the number of agents. The computing times and the number of agents are monitored in each

iteration during the simulation. This approach is executed by all the processes without a central unit of decision. Therefore, each process knows the global load situation and executes the algorithm with the same input. Consequently, all processes calculate the same reconfiguration of the workload. The schema is triggered when an imbalance factor is detected outside the tolerance range. This factor indicates the percentage of imbalance in respect to the mean, and the tolerance range defines the permissible degree of imbalance (see algorithm 3).

During the experiments, the load balancing schema is activated after the fifth simulation step. After each load balancing activation, the schema is disabled during one simulation step because the computing measurements vary when the migration process has been launched. The migration routines were tested using FLAME 0.16.2, libmboard 0.2.1 and OpenMPI 1.4.1. The experiments were executed on an IBM Cluster with the following features: 32 IBM x3550 Nodes, 2xDual-Core Intel(R) Xeon(R) CPU 5160 @ 3.00GHz 4MB L2 (2x2), 12 GB Fully Buffered DIMM 667 MHz, Hot-swap SAS Controller 160GB SATA Disk and Integrated dual Gigabit Ethernet. Tests consists of a case without load balancing schema, and three imbalance tolerances: 0.3(30%), 0.15(15%) and 0.05(5%).

The simulations were started with an initial population of 50.000 and 100000 agents (scenarios A and B, respectively), carrying capacity of 50.000 and 100.000, and 10 of space dimensions of 1.000x1.000 and 900x900, respectively. The environmental configuration of the simulations for both scenarios were set up as infected = 10, lifespan = 100, average offspring = 4, infectiousness = 65, chance recovery = 50, and disease duration = 20. Moreover, for both scenarios, 16, 32, 64 and 128 cores were used.

Given that the agents were distributed through a round-robin distribution, depending on the number of processes and the initial agents, the initial number of agents per process can be equal or similar. Figures 4(a) and 4(b) compare the execution time with a varying number of processes by comparing different values of tolerance with the original simulation without the load balancing schema.



(a) (b)

Fig. 4. Executions times in both scenarios

Table 1. Summarize of the execution of scenario B, 100.000 agents for 128 processes

tolerance	pack	unpack	agents-sent	kbytes-sent	exchanges	exec-time(sec)	gain(%)
-	-	-	-	-	-	2462,49	-
0,30	0,007	0,005	19985	937,38	119	2206,21	10,4
0,15	0,008	0,006	24080	1129,97	249	2048,76	18,8
0,05	0,010	0,007	37283	1751,89	870	1953,43	24,8

Here, both scenarios have better results using the load balancing schema. Moreover, in most cases, if the imbalance tolerance is reduced the improvement is better.

In Table 1, a reduced value of imbalance tolerance results in a better execution time. By contrast, achieving a better balance involves a larger amount of agents migrated and a larger amount of exchanges between processes. For this reason, a load balancing schema should consider the communication overhead caused by the amount of exchanges during the migration process.

5 Conclusion and Future Work

ABMS applications may present computational and communicational imbalances during the simulation process. Therefore, the simulation environment should be equipped with migration mechanisms to make modifications in the workload of the overloaded and underloaded processes.

In this paper, a modification of the FLAME framework for automatically generating agent migration functions is presented. Using the same FLAME code generation methodology, the migration routines are automatically generated from templates. Therefore, this improvement will allow the inclusion of load balancing strategies. In this manner, the workload among the different processes can be dynamically adjusted during the simulation.

It is planned to generalize a load balancing strategy and research on balancing communication times.

References

1. Allan, R.: Survey of agent based modelling and simulation tools. Engineering 501, 57–72 (2009)
2. Standish, R.K., Leow, R.: Ecolab: Agent based modeling for C++ programmers. CoRR cs.MA/0401026 (2004)
3. Collier, N., North, M.: Repast SC++: A platform for large-scale agent-based modeling. Large-Scale Computing Techniques for Complex System Simulations (2011)
4. 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)

5. Carillo, M., Cordasco, G., Chiara, R.D., Raia, F., Scarano, V., Serrapica, F.: Enhancing the performances of D-MASON—a motivating example. In: Pina, N., Kacprzyk, J., Obaidat, M.S. (eds.) SIMULTECH, pp. 137–143. SciTePress (2012)
6. Kiran, M., Richmond, P., Holcombe, M., Chin, L.S., Worth, D., Greenough, C.: FLAME: simulating large populations of agents on parallel hardware architectures. In: AAMAS, pp. 1633–1636 (2010)
7. Deissenberg, C., Vanderhoog, S., Dawid, H.: EURACE: A massively parallel agent-based model of the european economy. Applied Mathematics and Computation 204(2), 541–552 (2008)
8. Zhou, B., Zhou, S.: Parallel simulation of group behaviors. In: Proceedings of the 36th Conference on Winter Simulation, WSC 2004. Winter Simulation Conference, pp. 364–370 (2004)
9. Zheng, G., Meneses, E., Bhatele, A., Kale, L.V.: Hierarchical load balancing for Charm++ applications on large supercomputers. In: Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW 2010, pp. 436–444. IEEE Computer Society, Washington, DC (2010)
10. Cosenza, B., Cordasco, G., De Chiara, R., Scarano, V.: Distributed load balancing for parallel agent-based simulations. In: Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2011, pp. 62–69. IEEE Computer Society, Washington, DC (2011)
11. Zhang, D., Jiang, C., Li, S.: A fast adaptive load balancing method for parallel particle-based simulations. Simulation Modelling Practice and Theory 17(6), 1032–1042 (2009)
12. Vigueras, G., Lozano, M., Orduña, J.M.: Workload balancing in distributed crowd simulations: the partitioning method. J. Supercomput. 58(2), 261–269 (2011)
13. Solar, R., Suppi, R., Luque, E.: Proximity load balancing for distributed cluster-based individual-oriented fish school simulations. Procedia Computer Science 9, 328–337 (2012); Proceedings of the International Conference on Computational Science, ICCS 2012
14. Worth, D., Chin, S., Greenough, C.: FLAME tutorial examples: a simple SIR infection model. Technical Report RAL-TR-2012-017, Rutherford Appleton Laboratory (November 2012)
15. Márquez, C., César, E., Sorribes, J.: A load balancing schema for agent-based SPMD applications. In: PDPTA (2013)