MARL-Ped+Hitmap: Towards Improving Agent-based Simulations with Distributed Arrays

Eduardo Rodriguez-Gutiez[1], Francisco Martinez-Gil[2], Juan Manuel Orduña[2], and Arturo Gonzalez-Escribano[1] *

[1] Dpto. de Informática, Universidad de Valladolid,
Campus Miguel Delibes s/n, 47011 Valladolid (Spain),
{eduardo,arturo}@infor.uva.es
[2] Dpto. de Informática, Universidad de Valencia,
Avda. Universidad s/n, 46100 Burjassot (Valencia, Spain)
{francisco.martinez-gil,juan.orduna}@uv.es

Abstract. Multi-agent systems allow the modelling of complex, heterogeneous, and distributed systems in a realistic way. MARL-Ped is a multi-agent system tool, based on the MPI standard, for the simulation of different scenarios of pedestrians who autonomously learn the best behavior by Reinforcement Learning. MARL-Ped uses one MPI process for each agent by design, with a fixed fine-grain granularity. This requirement limits the performance of the simulations for a restricted number of processors that is lesser than the number of agents. On the other hand, Hitmap is a library to ease the programming of parallel applications based on distributed arrays. It includes abstractions for the automatic partition and mapping of arrays at runtime with arbitrary granularity, as well as functionalities to build flexible communication patterns that transparently adapt to the data partitions.

In this work, we present the methodology and techniques of granularity selection in Hitmap, applied to the simulations of agent systems. As a first approximation, we use the MARL-Ped multi-agent pedestrian simulation software as a case of study for intra-node cases. Hitmap allows to transparently map agents to processes, reducing oversubscription and intra-node communication overheads. The evaluation results show significant advantages when using Hitmap, increasing the flexibility, performance, and agent-number scalability for a fixed number of processing elements, allowing a better exploitation of isolated nodes.

Keywords: Agents, crowd simulation, message-passing, programming tools, distributed arrays

* This work has been funded by Spanish MINECO and the EU ERDF program under grants HomProg-HetSys TIN2014-58876-P, TIN2015-66972-C5-5-R, CAPAP-H5 network TIN2014-53522-REDT, and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

1 Introduction

Multi-agent systems allow the modelling of complex, heterogeneous, and distributed systems, in a realistic way. They assign an agent to each entity involved in the real-world environment [18, 17]. This software paradigm is particularly appropriated for the study of pedestrian dynamics, where autonomous interactions among individuals generate global system behaviors. MARL-Ped [13] is a multi-agent distributed tool where each agent (pedestrian) learns its own behavior by Reinforcement Learning (RL) [15], allowing the simulation of pedestrian groups (ranging from a few ones to crowds) in different scenarios (queue forwarding, congestion scenarios, evacuation of enclosed ares, etc.). The great computational workload added by the learning process of each agent, together with the required number of agents in medium and large scale scenarios require the use of High Performance Computing platforms. Indeed, the number of agents tested in learning environments is usually limited by the available computing resources. MARL-Ped is based on the MPI message-passing standard which provides portability across distributed- and shared-memory environments. It uses one MPI process for each agent by design, with a fixed fine-grain granularity. This requirement limits the performance of the simulations for a restricted number of processors that is lesser than the number of agents. On the other hand, Hitmap [7] is a library designed to ease the task of programming parallel applications by using distributed arrays. It includes abstractions for the automatic partitioning and mapping of arrays with arbitrary granularity, as well as the automatic construction of flexible communication patterns adapted to the partition.

In this work, we present the methodology and techniques of granularity selection in Hitmap applied to the simulations of agent systems, using MARL-Ped as a case of study. Hitmap allows to transparently map agents to processes. We show the benefits of using this mechanism for improving the performance of agent-based applications executed in a restricted number of processing elements that is lesser than the number of agents. It eliminates oversubscription effects, and reduces intra-node communication overheads by grouping communications. The application of the Hitmap methodology does not increase the development effort. The comparative performance evaluation shows that the version using Hitmap uses more efficiently the computing resources, becoming more scalable in terms of the number of simulated agents.

The rest of the paper is organized as follows: Section 2 shows some related work. Section 3 introduces MARL-Ped and Hitmap tools. Next, Section 4 describes how Hitmap has been included in the MARL-Ped original application. Then, Section 5 presents an experimental evaluation of the modified application. Finally, Section 6 discusses some conclusion remarks and future work to be done.

2 Related Work

Pedestrian-dynamics models were improved and extended in the 80s with the advent of low cost computers. Many different models have been used: the social

forces model [9], models based on cellular automata [2], or continuum models based on gas kinetics equations [10]. However, the most extended ones are agent-based models [14], due to the ease of extracting global behavior as the sum of individual behaviors. In the last years, some efforts have been made to add machine learning technique to agent-based pedestrian models [12], in such a way that the agents learn their individual behavior by themselves, releasing the programmer of this task. Since the behavior learning is a complex task, it has become the main challenge for the pedestrian models. On the other hand, the microscopic simulation of pedestrian in crowded scenarios requires parallel processing. In this sense, specific architectures have been proposed for these simulations [1], and parallel architectures, where interconnected servers share the computational workload, have been developed [16]. Even architectures based on many-core processors have been used for simulating a marathon of one million runners [19].

Hitmap offers an intermediate abstraction layer, halfway between the manual programming of distributed data structures on message-passing models, and PGAS languages (Partitioned Global Address Space), like Chapel [3] or UPC [11]. Hitmap also provides mechanisms for the construction of reusable communication patterns at runtime that adapt to the data partition, creating a low number of aggregated communications. This leads, for example, to a performance efficiency comparable to UPC, with a reduced programming complexity and development effort [7]. Hitmap is used as a runtime system for the Trasgo parallel programming framework [8], that offers an approach similar to PGAS languages. Hitmap extends and generalizes the hierarchy creation and data partition functionalities of other libraries or distributed arrays models, such as HTAs [5] o Parray [4]. It allows to use transparent partition policies, either regular or irregular, defined as interchangeable modules with a common interface. This hides to the programmer the decisions about granularity and synchronization across hierarchical levels. Hitmap has also been extended to support data structures such as sparse matrices, or graphs, using the same methodology and interface [6].

3 MARL-Ped & Hitmap

3.1 MARL-Ped

MARL-Ped is a multi-agent system tool for pedestrian simulation which uses reinforcement learning (RL) [15] in each agent to learn the individual behavior of a single pedestrian. The purpose of the RL algorithm is to compute a control function which will be used by the agent to select at a given moment the action to do, based on the sensorized local state. MARL-Ped includes two types of agents: (a) Pedestrian (Learning) agents, which execute the RL algorithms and store the control function learned; and (b) an Environment agent, which execute the physical system simulation of the scenario, and sensorizes the state of each agent. The scenario is a 3D virtual world where the physical model engine named Open Dynamic Engine (ODE) simulates the collisions and forces moving the

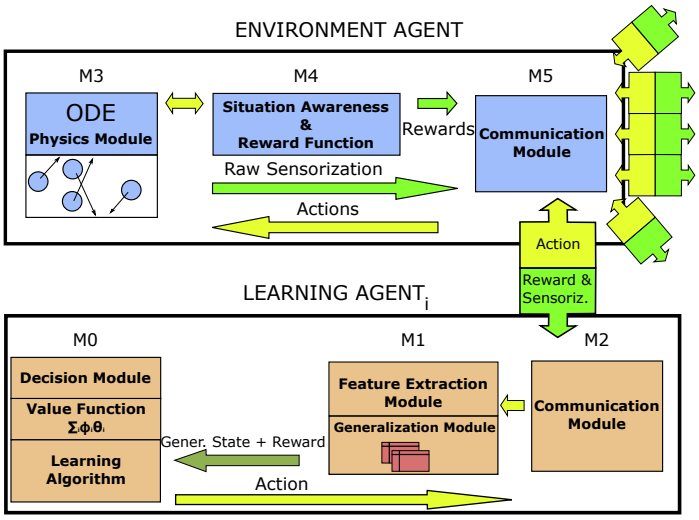

Fig. 1. MARL-Ped scheme showing the types of agents and their relationships.

pedestrians. Fig. 1 shows a graphic scheme of the system, including both types of agents, and the communications exchange. These communications take place exclusively between the Environment agent and the rest of agents.

MARL-Ped has two working modes: learning mode and simulation mode. Both modes include the same communications between learning agents and the environment. The only difference is that RL algorithms are active in the learning mode to incrementally compute the control function, that will be used in the simulation mode. Both modes are synchronous, and composed of the classical cycle of observation-action-reward:

1. The Environment agent queries the ODE about the dynamic situation of each agent, consisting of position, speed, distance to the closest n pedestrians, and the distance to the closest n objects. In learning mode, the Environment agent also assigns a reward for each pedestrian agent depending on different facts: if it has reached the target, if it has collisioned with other agents or objects, etc.
2. The Environment agent sends the state and reward information to the Learning agents.
3. Each Learning agent uses the received information to build the local state and the immediate reward value. In the learning mode, the data built will be used by the RL algorithm to update the control function. In the simulation mode, the control function is not updated.
4. The agent queries the current control function to obtain the new action to be executed. The action indicates a change in direction and/or speed of the pedestrian.

5. The agents send their actions to the Environment agent, which in turn translate them into physical actions executed by the ODE in the virtual environment.

This cycle is repeated a given number of times which is a configuration parameter of the system. In the learning mode with some tens of agents, this parameter can range from hundreds of thousands to several million times.

3.2 Hitmap

Hitmap [7] is a library for the partition, mapping, and management of hierarchically distributed data structures at runtime. It was originally designed for dense arrays, and has been also extended to support sparse data structures, such as sparse matrices or graphs, using the same methodology and interface [6]. It is based on an SPMD (Single Program Multiple Data) model and the message-passing paradigm. Hitmap defines several abstractions to write parallel programs using distributed data structures. The functions in the library are grouped in three main modules.

Tiling functions. They allow the definition and management of hierarchically tiled data structures. These functionalities can be used independently of the rest of the library to improve locality on sequential code. They define classes to represent domains of indexes in a compact form. A class named *HitTile* represents the association between the elements of the indexes-domain space and the actual data, allowing the accesses to data with the same efficiency as manually developed codes without the tile abstraction. A process can declare and allocate a subspace of the original domain, in order to create a distributed data structure.

Mapping functions. They include interchangeable modules that implement policies to automatically part and map domains in terms of the processes of a virtual topology. The virtual topologies are also generated by another class of policy modules at runtime. Neighbor relations across processes are established by these policies. The partitions are represented by objects named *HitLayouts* that can be queried to obtain the indexes subdomain mapped to the local, a neighbor, or any other remote virtual process.

Communication functions. They are an abstraction of the message-passing model for tiles or tiles parts across virtual processes. They allow the creation of *HitCom* objects that store the information needed to marshall/unmarshall and exchange selected tile data across processes. Several interfaces for different types of point-to-point and collective communications are available. More complex patterns composed of multiple communication operations involving one or more tiles (several HitCom objects), are implemented as *HitPattern* objects. The constructor functions have always *HitLayout* parameters that are queried internally to automatically determine who communicates and what. Thus, these objects are transparently adapted on construction to the target platform details and the actual data distribution selected. The communication objects have a method that can be called at any time, and as many times as needed, to execute the communications. Internally, these objects exploit efficient MPI techniques such as derived data types, asynchronous communications, etc.

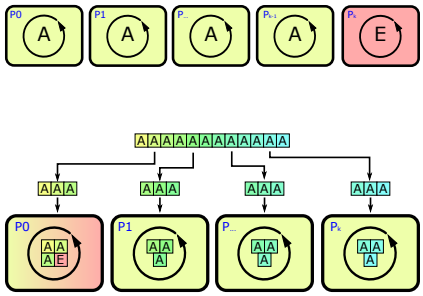

Fig. 2. Global structure of the simulation and task distribution across processors in the original MARL-Ped design (top) and after applying Hitmap (bottom).

4 Applying Hitmap Techniques & Methodology

In this section we describe how the Hitmap methodology and techniques can be applied to agent-based simulation applications to adapt the granularity of tasks to the available processing resources. We show this process using MARL-Ped as a case of study.

4.1 Structural Changes

The structure of the MARL-Ped application has been redesigned. The Hitmap version applies the concept of distributed arrays to group learning agents in processes, instead of using a single MPI process for each one, and a different process for the environment agent. Fig. 2 (top) shows the conceptual distribution of the computation in the original MARL-Ped version. Each process executes the code of a single agent (RLAgent class). The last process performs the environment simulation (RLEnvironment class). The objects of these classes have several methods that implement the corresponding operations of the simulation loop that is repeatedly executed.

One of the first design decisions for the Hitmap version is to distribute agents across the available processes without reserving a special process for the environment. The environment code will be executed by one of the processes that will also have learning agents assigned, as the main computation for the learning agents and environment never overlap in time. Hitmap provides the tools needed for the balanced distribution of agents between the available processes as depicted in Fig. 2 (bottom). Each process should be able to execute, for each iteration of the simulation loop, the code of several learning agents.

In addition, the process in which the environment agent is mapped should execute its code. Thus, the simulation loop code cannot be placed inside the environment or learning agent classes. The application must be redesigned to execute the simulation loop in the main function. The simulation loop must iterate across the number of agents mapped to the process. To achieve this, the

codes of the simulation loop are removed from the methods of the learning and environment classes. The private and protected methods called inside the loops are redeclared as public. The control logic that do the calls is relocated inside the new simulation loop at the main function. The environment control logic is wrapped with conditionals to ensure that only one process executes it. Hitmap automatically labels one process as the group leader. This process can identify itself by using a function call, and is therefore the one selected to execute the environment logic.

4.2 Distributed Arrays and Communication Patterns

The MPI-based communications in original MARL-Ped code have been replaced by distributed-array management functions provided by Hitmap. All data structures involved in communications are substituted by *HitTile* structures.

During the initialization stage of the program, the distributed arrays and objects of type *HitCom* and *HitPattern* are created to contain the specifications of the communications that will be invoked from the new simulation loop. Control signals are represented by a single integer-type variable at each process, independently of the number of assigned agents. On the other hand, two distributed arrays are declared for each data flow between the environment and the learning agents. These arrays have a global index domain equal to the number of learning agents. For one of the arrays, we use a distribution policy that maps its elements evenly across the processes. For the other one, we use a policy that maps all of the domain elements to the process running the environment. Given these two arrays with the same domain but different distribution policies, Hitmap allows the creation of a *HitPattern* object with a single function call. This object implements a communication pattern capable of redistributing the data from one array to the correspondent local or remote elements of the other array. This technique allows the construction of communication objects that will transparently move the data between the two copies of each array; the one actually distributed and the other one having the entire index domain at the environment process. The communication pattern adapts (at construction time) to the results of the partition policies, regardless of the number of agents and processes. This mechanism solves, in a unique way, the construction of the communication flows.

5 Experimental Study

This section describes an experimental study to show the advantages of using Hitmap on agent-based simulation programs. The study is focused on two areas. The first one is the code complexity and development effort. The second one is the performance when the number of agents grows above the number of available processing elements.

	MARL-Ped	MARL-Ped+Hitmap
KDSI (code lines)	1970	1888
McCabe's C.C.	209	171
Halstead	19.38×10^6	18.26×10^6

Table 1. Measurements of complexity and development effort.

5.1 Development Effort

The first part of this experimental study shows that programming with Hitmap introduces granularity flexibility, even with a slightly lower development effort and code complexity than the original agent-per-process approach. We have measured several metrics both in the original MARL-Ped source code and in the modified Hitmap version: (a) The KDSI metric of the COCOMO methodology, based in the total number of source code lines; (b) McCabe's cyclomatic complexity; and (c) Halstead development effort metric. We have applied these metrics on the main function of the programs and the three classes modified when redesigning the original application. We have considered both the code of the modified functions and the header files, excluding comments and removing conditional compilation parts related to versions, alternatives or details of the MPI libraries used, etc. The modified code represents 16% of the total application code, that has approximately 12 200 lines of code.

The results in Table 1 show that the version directly designed and programmed using Hitmap presents slightly lower complexity and effort than the original MPI version. Programming a direct MPI version with the agent distribution and load balancing capacity of the Hitmap version would clearly increase the programming effort, since the programmer would have to include code dealing with decisions about distributed array partition and management, that are transparently implemented in Hitmap.

5.2 Experimental Methodology for Performance Studies

The second part of the experimental study includes performance measurements of both the original MARL-Ped program and the Hitmap-based version. This work is focused on the MARL-Ped learning process, which is the most computationally demanding mode, and does not imply input/output operations during the main computation and communication loop. The code has been instrumented in order to measure the execution time for each distributed process. We have measured the time elapsed from the start of the initialization of parallelism-related structures (MPI or Hitmap) to the end of the execution of the learning process, before writing the results in files. Since each execution of the whole program gives one time measurement for each process, we consider as the global result the time of the slowest process, the one that has required the longer time to be completed. In addition, each experiment has been repeated several times in

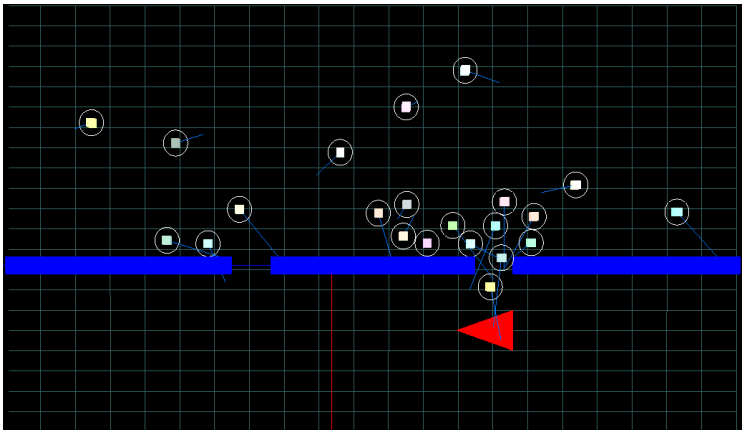

Fig. 3. Snapshot of the simulated scenario.

order to test the variability of the results. Both codes have been executed in multicore platforms, where communication costs are lower and potential overheads have a higher impact on the overall performance. These potential overheads can be associated to changes in execution structure, handling of internal Hitmap data structures, or computations and choices about the particular communications, among others. We have selected two machines, one with 8 cores (named *Miami*), and the other with 12 cores (named *Chimera*). Both machines had the *hyperthreading* option enabled. Table 2 summarizes the characteristics of these platforms as well as the development tools used in the study.

Since the execution time required for a full learning process execution is extremely long (RL is based on a long iterative process), the program has been limited to only 100 training iterations in all cases, in order to analyze a search space that is broad enough in terms of execution parameters. This threshold has been experimentally set to produce both a large computational load, and a significant number of communication and synchronization steps. The test scenario selected for the experiments has been validated in previous works [13]. This scenario reproduces a classic navigation problem in pedestrian dynamics called "shortest path vs. quickest path". In this scenario, a group of pedestrians must move from the room where they are initially located to a target place located outside of the room. This room has two exits, one of them being closer to the target than the other one. Agents must learn that if all of them head for the nearest exit, then a bottleneck is formed, making the overall evacuation time longer. A better solution implies that approximately half of the agents use the nearest exit, while the other half leaves the room through the most distant one, leading to a quicker evacuation. The configuration chosen places 28 agents in a 30-meter by 30-meter square room with two possible exits. Each exit has a width of one meter in order to prevent passage of more than one pedestrian

at the same time. The goal of the agents is to reach the meeting point placed outside the room. Fig. 3 illustrates the considered scenario, showing a snapshot of the simulation.

5.3 Performance Effect of the Agents Grouping

One of the objectives of this work is to obtain a better scalability when the number of agents grow, for a restricted number of processing elements in single cluster nodes. It is achieved using the Hitmap strategy of grouping several agents to the same process. This first performance study experimentally tests the difference in performance when the number of agents increases above the number of available processing elements in a given machine. Both programs (the original MARL-Ped version and the Hitmap-based version) have been run in learning mode with the number of MPI processes set to the number of cores available on each machine, and we have progressively increased the number of agents above that number.

Fig. 4 shows the execution times required by *Miami* and *Chimera* machines (using 8 and 12 MPI processes respectively) for simulations of 100 learning iterations with different numbers of learning agents. In the original MARL-Ped tool (whose plots are labeled as MARL-Ped in the figure), the number of MPI processes is the number of agents plus one. In the MARL-Ped+Hitmap tool (whose plots are labeled as Hitmap in the figure), the number of processes has been fixed to be the number of actual cores of the machine. The structure of these simulation applications, that make use of collective communications with clear global synchronization points around the execution of the simulation engine, does not present a *parallel slackness* property. This property appears in certain applications when several processes assigned to the same element alternate communication and computation phases without overlapping.

Fig. 4 shows a very similar behavior in both platforms: the required execution time is slightly longer for MARL-Ped than for the Hitmap version while

		Miami	Chimera
Processor		2x Intel X5550	2x Intel E5-2620 v2
Clock speed		2.66 GHz	2.10GHz
Cores		8	12
Main memory		32GB DDR3-1333	8GB DDR4-1866
Cache	L1	128K	32K
	L2	1024K	256K
	L3	8192K	15360K
Operative system		CentOS 7.2.1511 x64	CentOS 7.0.1406 x64
C++ Compiler		GCC 4.8.5 20150623	GCC 4.8.2 20140120
Compilation flags		-O3	-O3
MPI implementation		MPICH 3.0.4	MPICH 3.1.3

Table 2. Characteristics of the machines used in the experimental study.

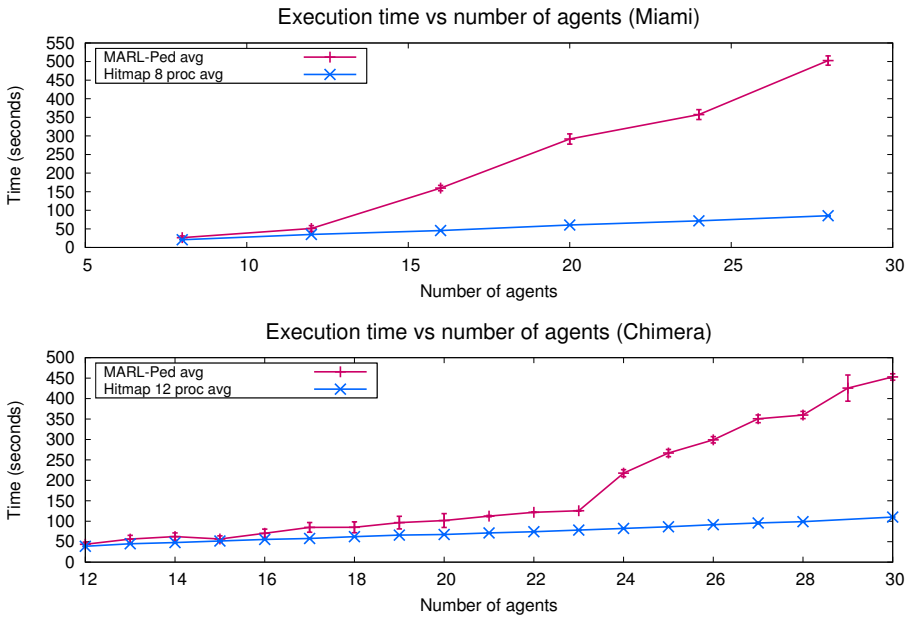

Fig. 4. Execution time vs. number of agents for 100 iterations of learning. The plots show with error bars the standard deviation of six different executions.

the number of agents does not reach double the number of existing cores in the machine. From this value up, the execution time required by MARL-Ped is significantly higher, linearly increasing with a high slope. These plots show how the additional costs of *oversubscription* (executing simultaneously more processes than processing elements available) has a negative impact on the application performance. Thus, in the Chimera machine, with 12 cores and hyperthreading enabled, the effect is much more remarkable starting at 24 agents, where MARL-Ped uses 25 MPI processes, increasing the oversubscription ratio to more than two. When there is oversubscription, the way in which task schedulers swap processes becomes relevant. Thus, we can observe slightly higher differences between maximum and minimum execution times in both *Miami* and *Chimera* machines for the original MARL-Ped program. On the contrary, the Hitmap plots remain almost flat regardless of the number of agents, showing much shorter execution times than the original MARL-Ped. This behavior is due to the fact that Hitmap can limit the number of processes, executing sequentially on each process the code of several agents in a more efficient way.

5.4 Impact of the Amount of Processes

This section studies the impact of modifying the number of processes, and therefore the agents distribution per process, when MARL-Ped+Hitmap is used.

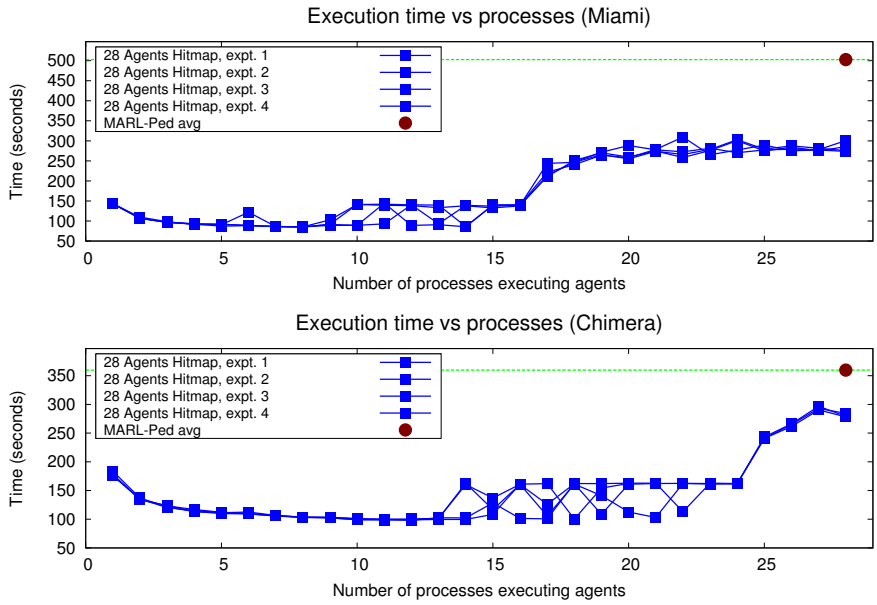

Fig. 5. Execution time vs. number of processes for 100 iterations of training, with a fixed number of 28 agents.

Fig. 5 shows the results obtained for a fixed number of 28 agents in the two machines. For MARL-Ped+Hitmap, we include plots for several consecutive experiments to show the variability of the results. The execution results of MARL-Ped+Hitmap are better than those of the original MARL-Ped in all cases, because of the oversubscription effects discussed in the previous section. It can be seen that the results for the Hitmap version when executed with a single process are slightly improved when the number of processes grows up to the number of cores in the machine, since the parallelism level increases. However, when the number of processes exceeds the number of real cores (without taking hyperthreading into account), the performance decreases and results become more unstable, due to stochastic negative effects of oversubscription. The process scheduling policy of the operative system also contributes to make the results more unpredictable. Due to the execution order of the processes during the context switching, in some cases execution times are as short (good) as before the start of the oversubscription, while in other cases the results are worse, but with a clear upper limit. Once the number of processes exceeds the number or available threads, taking hyperthreading into account, the results considerably worsen.

These results indicate that the number of processes to select in MARL-Ped+Hitmap in training mode is predictable, and can be adapted to the features of the target machines when the application is launched. The execution times

are shorter and more stable when the number of processes is equal to the number of real processing elements, without taking into account the hyperthreading option.

6 Conclusions

This article presents the application of the techniques and tools of the Hitmap library to control the granularity of agent to processes map in agent-based simulation applications. As a first approximation, we use the MARL-Ped multi-agent pedestrian simulation software as a case of study for intra-node cases. The performance evaluation results show that MARL-Ped+Hitmap allows simulations with a number of agents greater than the number of processing elements available in a machine, while keeping execution times stable and predictable, These results show that the use of distributed arrays and automatic data partitions improves the performance agent-based simulation tools, due to the ability of Hitmap of transparently map a high number of agents to a constricted number of processes.

Future work includes the study of the scalability of Hitmap techniques in multi-node clusters when the effects of network and communication across nodes appear; the application of Hitmap to other related simulation applications; research on ways to suppress or mitigate bottlenecks in the simulation stages; and the use of the new MARL-Ped+Hitmap version to further study the quality and results of crowd simulations with a much greater number of agents.

References

1. Bharambe, A., Pang, J., Seshan, S.: Colyseus: a distributed architecture for online multiplayer games. In: NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation. pp. 12–12. USENIX Association, Berkeley, CA, USA (2006)
2. Blue, V.J., Adler, J.L.: Cellular automata microsimulation for modeling bi-directional pedestrian walkways. Transportation Research Part B: Methodological 35(3), 293–312 (2001)
3. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. 21(3), 291–312 (aug 2007)
4. Chen, Y., Cui, X., Mei, H.: Parray: A unifying array representation for heterogeneous parallelism. SIGPLAN Not. 47(8), 171–180 (feb 2012)
5. Fraguela, B.B., Bikshandi, G., Guo, J., GarzaráN, M.J., Padua, D., Von Praun, C.: Optimization techniques for efficient hta programs. Parallel Comput. 38(9), 465–484 (sep 2012)
6. Fresno, J., Gonzalez-Escribano, A., Llanos, D.: Blending extensibility and performance in dense and sparse parallel data management. IEEE Transactions on Parallel and Distributed Systems 25(10), 2509–2519 (2014)
7. Gonzalez-Escribano, A., Torres, Y., Fresno, J., Llanos, D.: An extensible system for multilevel automatic data partition and mapping. IEEE Transactions on Parallel and Distributed Systems 25(5), 1145–1154 (2014)

8. Gonzalez-Escribano, A., Llanos, D.R.: Trasgo: a nested-parallel programming system. The Journal of Supercomputing 58(2), 226–234 (2011)
9. Helbing, D., Molnár, P.: Social force model for pedestrian dynamics. Phys. Rev. E 51, 4282–4286 (1995)
10. Hughes, R.L.: The flow of human crowds. Annu. Rev. Fluid Mech. 35, 169–182 (2003)
11. Mallón, D.A., Gómez, A., Mouriño, J.C., Taboada, G.L., Teijeiro, C., Touriño, J., Fraguela, B.B., Doallo, R., Wibecan, B.: Upc performance evaluation on a multicore system. In: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models. pp. 9:1–9:7. PGAS '09, ACM, New York, NY, USA (2009)
12. Martinez-Gil, F., Lozano, M., Fernández, F.: Multi-agent reinforcement learning for simulating pedestrian navigation. In: Adaptive and Learning Agents: International Workshop, ALA 2011, Held at AAMAS 2011, Taipei, Taiwan. pp. 54–69. Springer Berlin Heidelberg (2012)
13. Martinez-Gil, F., Lozano, M., Fernández, F.: MARL-ped: A multi-agent reinforcement learning based framework to simulate pedestrian groups. Simulation Modelling Practice and Theory 47, 259–275 (2014)
14. Reynolds, C.: Steering behaviors for autonomous characters. In: Game Developers Conference. pp. 763–782. Miller Freeman Game Group, San Francisco, California. (1999)
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)
16. Vigueras, G., Orduña, J.M., Lozano, M.: A read-copy update based parallel server for distributed crowd simulations. The Journal of Supercomputing 64(1), 156–166 (2013)
17. Wooldridge, M.: Multi-Agent Systems 2nd Edition, chap. Intelligent Agents, pp. 3–50. MIT Press (2013)
18. Wooldridge, M., Jennings, N.: Intelligent agents: theory and practice. The Knowledge Engineering Review 10, 115–152 (1995)
19. Yilmaz, E., Isler, V., Cetin, Y.Y.: The virtual marathon: Parallel computing supports crowd simulations. IEEE Computer Graphics and Applications 29(4), 26–33 (2009)