Load-Sharing Policies in Parallel Simulation of Agent-Based Demographic Models

Alessandro Pellegrini[1], Cristina Montañola-Sales[2], Francesco Quaglia[1], and Josep Casanovas-García[2]

[1] DIAG, Sapienza, University of Rome
{pellegrini,quaglia}@dis.uniroma1.it
[2] inLab FIB, Barcelona School of Informatics
cristina.montanola@upc.edu, josepk@fib.upc.edu

Abstract. Execution parallelism in agent-Based Simulation (ABS) allows to deal with complex/large-scale models. This raises the need for runtime environments able to fully exploit hardware parallelism, while jointly offering ABS-suited programming abstractions. In this paper, we target last-generation Parallel Discrete Event Simulation (PDES) platforms for multicore systems. We discuss a programming model to support both implicit (in-place access) and explicit (message passing) interactions across concurrent Logical Processes (LPs). We discuss different load-sharing policies combining event rate and implicit/explicit LPs' interactions. We present a performance study conducted on a synthetic test case, representative of a class of agent-based models.

1 Introduction

Agent-based modeling (ABM) is a simulation technique which provides abstract representations of a scenario via a descriptive model to reproduce its evolution through its components, including their decision-making capabilities and interaction patterns. An agent can be defined as an entity (theoretical, virtual or physical) capable of acting on itself, on the environment in which it evolves, and capable of interacting with other agents [1]. ABM is very useful in capturing interactions at a macro scale coming from the way agents behave at a micro level. This intrinsic expressive power makes it a proven solution to explore complex real-world scenarios, such as disaster rescue [2], ancient societies resilience [3], epidemiology [4], and economic analysis [5].

Supporting the execution of simulation models expressed using such a versatile formalism is a task which requires a substantial methodological effort. In fact, a large number of widely-adopted ABM frameworks [6–9] is intrinsically serial, and can therefore handle a *population* which is significantly limited in its size. To avoid limiting the speed and scalability of simulations, efficient parallelization techniques must be employed. On this trend, several works aim at exploiting the high parallelism offered by GPU computing [10, 11] or cluster-based parallel computing [12]. More in general, Discrete Event Simulation (DES) can be considered as a mainstream formalism to describe agent-based models. The reason

is that agents' interactions can be abstracted as occurring at particular time instants—interactions having a specific duration can be mapped to a couple of *begin* and *end* discrete events. The mapping from ABM to DES is trivial, as the entities (agents and the environment) can be easily mapped to the general notion of Logical Process (LP) proper of DES. This is an important aspect, given the existence of a plethora of techniques globally referred to as Parallel Discrete Event Simulation (PDES) [13], which provide protocols and mechanisms to run complex DES models in parallel, allowing for model speedup and tractability of more complex and large models.

In this paper, we discuss a reference programming model for agent-based demographic models to be run on top of shared-memory PDES systems. In particular, we target the speculative paradigm incarnated by the well-known Time Warp synchronization protocol [14], which has been recently shown to provide scalability up to thousands or millions of CPU-cores [15]. Our goal is to give the highest degree of freedom to the programmer, and to ensure an efficient execution of the simulation. We target symmetric-multithread PDES environments for shared-memory multicore systems [16]. LPs are allowed to interact in a twofold way: (i) *explicitly*, namely via traditional message passing, or (ii) *implicitly*, i.e. relying on in-place memory accesses of their respective simulation states. This latter interaction is based on *cross-state synchronization* [17] to track memory areas accessed by threads scheduling LPs, and it has already been proven a good facility to enhance the programmability of agent-based models [18].

Moreover, we present three *load-sharing policies* to optimize the *binding* between LPs and worker threads of the simulation platform. As discussed in [16], this is a fundamental aspect to offer competitive performance. The binding temporarily assigns computing resources (i.e., worker threads stick to certain cores) to groups of LPs. Their composition can significantly affect the overall performance due to, e.g., reduced rollback frequency. The policies are based on: (i) *density of events* in the future event list of LPs; (ii) implicit interactions among different LPs; (iii) both implicit and explicit interactions among the LPs. Different simulative scenarios can benefit from these policies, depending on the events' generation pattern and/or the amount and the nature of the interactions. A synthetic benchmark, representative of a wide range of ABM demographic models, is used to study these policies under different workload scenarios.

The load-sharing policies can bias synchronization dynamics to let a Time Warp system improve its performance when different portions of the simulation model exhibit stricter interdependencies. This can improve the usage of computing resources while carrying out speculative processing of DES models, by reducing negative effects of speculation, such as the rollback frequency. This is the objective of classical load balanging/sharing approaches proposed in literature (see, e.g., [19–21]). However, these proposal consider only explicit interactions supported via the classical event cross-scheduling approach.

The remainder of this paper is structured as follows. Section 2 discusses related work. In Section 3 we present our programming model. Section 4 introduces our load-sharing policies. The experimental assessment is provided in Section 5.

2 Related Work

In the literature, there are several frameworks to efficiently support agent-based simulation, both on distributed and shared-memory systems, or on GPUs.

The MASON framework [8] pays special attention to the performance of simulation execution, addressing computing-intensive models (i.e., large scenarios with many agents), along with portability and reproducibility of the results across different hardware architectures. A parallel/distributed version (D-MASON) has been presented in [12], which relies on time-stepped synchronization and on the master/slave paradigm. We similarly address the performance of agent-based simulation execution, yet we do this for the case of speculative asynchronous (non-time-stepped) PDES, reducing the negative effects of optimism by finding proper binding between LPs and threads.

AnyLogic [22] is a commercial multi-method general-purpose simulation modeling and execution framework, offering at the same time the possibility to support discrete-event, system dynamics, and agent-based simulation. The simulation model developer can rely on graphical modeling languages to implement the simulation models, along with Java code. Differently from this framework, we target C technology and explicitly provide self-tuning capabilities in the load-sharing policies, which allow to optimize at runtime the simulation performance.

FLAME [23] is a simulation framework targeting large, complex models with large agent populations to be run on HPC platforms using MPI and OpenMP. The counterpart FLAME GPU [24] targets 3D simulations of complex systems with a multi-massive number of agents on GPU devices. We keep the ability to deal with large amount of agents (bound by the simulation state size), yet we rely on traditional CPU-based execution of the simulation model.

In the context of PDES, several works have studied the problem of finding the best binding between LPs and worker threads—see, e.g., [16, 25, 26, 21, 27]. Nevertheless, none of these works has ever used information related to the interaction between LPs to explicitly reduce the (possible) negative effects of optimistic simulation runs.

The proposal in [28], still targeting multi-core architectures as we do, proposes a technique called Dynamic Local Time Window Estimates (DLTWE), in which each processor communicates time estimates of its next inter-processor event to its neighbors, which use the estimates as bounds for advancement. The proposal specifically targets spatial simulations, in which different (close) sub-volumes could be interested by a rollback operation. A selective rollback function is described, which allows to reduce the effects of rollbacks at LPs managing "close" entities. Contrarily, we do not impose any topology or predetermined relation across the LPs, which is an implicit outcome thanks to the different supported programming model (based on in-place state access everywhere). Moreover, we limit the effect of a rollback too for applications exploiting such a programming model by explicitly avoiding causal inconsistencies across LPs that are dynamically granulated together.

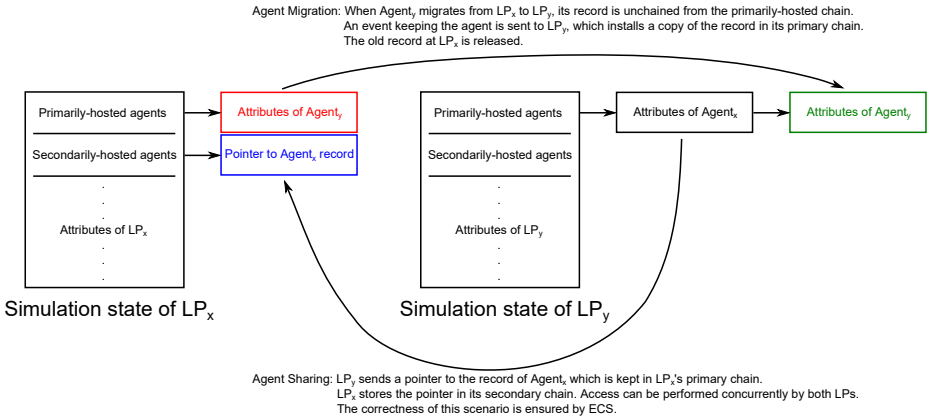


Fig. 1. Cross state-enabled programming model for agent-based demographic models.

3 Reference Programming Model

In the most general case, the core element of a demographic model is the life course of *individuals*, while their behaviour and their decisions strongly depend on the *environment* they act into [29]. ABM is interesting for demography due to its ability to generate personal-event histories and to produce estimates of the full distribution outcome [30]. Only two elements are required by any demographic agent-based model: the environment and the agents (with their interactions). Borrowing from the discussion in [31], we map environment portions or places to LPs, and agents to specific data structures managed by LPs' handlers.

Indeed, an agent can be described in terms of *individual-specific explanatory variables*. Changes in its state can be expressed as transitions (implemented within the LPs' event handlers) on some variables. In this way, different LPs can manipulate the same agent differently, giving more expressive power at no additional cost. The movement of an agent from a portion of space to another can be encoded by having the origin LP schedule an event carrying the agent's data structure(s) at the destination LP. This LP can then register the agent's records within its simulation state. A LP might implement any logic within its event handlers, and can therefore access any agent currently registered at it.

Nothing prevents multiple LPs from keeping in their states the records of the same agents. This reflects a scenario where LPs represent non-disjoint places, e.g., one LP might logically represent a city, while another LP might represent a workplace within it. Both LPs can manage a subset of the state transitions which involve an agent, and this organization clearly simplifies the implementation of the model, allowing for reuse/interoperability of different models. In this scenario, cross-state synchronization [17] becomes a mandatory aspect to deal with the correctness of the parallel simulation run.

By relying on cross-state synchronization, we can schematize our programming model for demography as in Figure 1. Each LP can describe a geographical

region or a specific place (e.g., a workplace or a hospital) within one of the geographical regions. Both kinds of LPs keep two lists of records, a *primary list* and a *secondary list*. The primary list keeps track of the agents currently in the region represented by the LP, and therefore the LP's handlers can manipulate their attributes. Each agent is identified by a system-wide unique id, so that a LP's hanlder can manipulate subsets of the currently-hosted agents. Similarly, the secondary list keeps track of the agents which can be managed (in terms of record update) by the LP, yet are not *primarily hosted* at the region. This is a list of *pointers* to some agent records kept in the primary list of any other LP in the system. In this way, multiple LPs share a portion of their simulation state, and concurrently access the records of the agents of interest for the execution of the model, decoupling different logical aspects of the model. For example, if a LP represents a workplace, all agents working there could have their salary updated via a simple chain traversal—this operation is independent of any other action involving the agents, and is thus realized on a separate module of the model.

By this organization of the LPs' states, we envisage two different operations on agents which are of general usability for demographic agent-based models:

- *Agent sharing*: if a LP wants to share an agent with other LPs, it simply sends an event carrying a pointer to the record chained to its primary list.
- *Agent migration*: when an agent physically moves from one spatial region to another, the source LP creates a copy of the agent's record into a message, which is scheduled to the destination LP with a model-specific timestamp increment. The record currently chained to the origin LP's primary list is detached and `free()`'d[1], and all the LPs keeping a pointer to the record are instructed via message passing to removed pointers from the secondary list.

If two agents want to interact, this is likely due to them being registered at the same LP (or shared across the same LPs), and their records can be easily retrieved from LPs' lists. In the more unlikely case that two agents interact *remotely* (e.g., they interact due to some kinship relation), this can be supported via traditional message passing. To this end, the source agent (run by its hosting LP) can keep the id of the destination LP within its record. The model should only ensure that when an agent migrates to another region, it informs (via message passing) the interested agents of their migration.

4 Load-Sharing Policies

4.1 Policy 1—Future Event List and GVT Advancement

The first policy we propose relies on a consensus algorithm to maximize the global event rate (namely, the global amount of committed simulation events per wall-clock-time unit) across all the worker threads. We consider the availability of C cores, and complying with the organization in [16], we assume K worker

[1] This pattern is compliant with traditional PDES environments, in which the virtual address of a buffer identifies its ownership with respect to a certain LP.

threads ($K \leq C$) are available for event processing. To determine what LPs should be bound to the available worker threads, we follow these steps:
Step 1. Each worker thread k_i, $i \in [1, K]$, hosts a set of LPs with cardinality $numLP^{k_i}$. We associate each LP_l, $l \in [1, numLP^{k_i}]$, with a *workload factor* L_l, defined as the wall-clock time needed to advance LP_l's local virtual time of one unit. The factor L_l is computed considering the number of events registered into the LP's future event list which fall within a distance in the future equal to the last GVT advancement normalized to the local virtual time advancement they would produce, weighted by the average CPU time for event processing by LP_l, that is:

$$L_l = \frac{q_l \cdot \delta_l}{LVT_l^{q_l} - LVT_l^1} \tag{1}$$

where q_l is the amount of events falling within the interval of interest, LVT_l^i is the timestamp of the i-th pending event in the queue, and δ_l is the average CPU time requirement for event processing by LP_j. Among the above parameters, q_l and LVT_l^i are known in advance, since they depend on the state of the input queue. Instead, δ_l is unknown since it expresses the average cost for events that have not yet been processed. Anyhow, it can be approximated by an exponential mean over already-processed events.
Step 2. The worker thread k_i computes its total workload as:

$$L^{k_i} = \sum_{l=1}^{numLP^{k_i}} L_l \tag{2}$$

Step 3. The actual bindings are determined, accounting for the highest workload factor found among LPs. This is done in several sub-steps based on *knapsack*:

- Workload factors for the LPs hosted by k_i are non-increasingly ordered (let us call them in this order as $L_{l_1}, L_{l_2}, \ldots, L_{l_H}$);
- The highest factor L_{l_1} is taken as the reference value, and the knapsack formed by LP_{l_1} is defined;
- The other knapsacks are built by aggregating the remaining LPs according to a *0-1 one-dimensional multiple knapsack* problem-solving algorithm. This problem is NP-hard, whose integral solution is non-trivial. So we rely on a *greedy approximation approach* [32], considering K knapsacks. At each step of the algorithm, $\forall i \in [2, H]$, the k-th knapsack's size is updated as $S_k = S_k + L_{l_i}$, and it is considered full if the size constraint is violated. We then switch to the $k+1$ knapsack, and begin to fill it. Once all K knapsacks are full, the remainder LPs (if any) are distributed in a round-robin fashion.

4.2 Policy 2—Implicit Synchronization

The memory management architecture in [17], allows to materialize cross-state accesses by leveraging a Linux kernel module which installs sibling page tables in x86 MMU registers. In this way, whenever a LP accesses a memory page bound to another LP, we can determine a cross-LP relation which we use to rebind

Algorithm 1 LP Grouping

```
1:  procedure REGROUP(LpGroup GLP, int LPid, int group)
2:      if GLP[LPid].group ≠⊥ then
3:          return GLP[LPid].group
4:      end if
5:      if group ≠⊥ then
6:          GLP[LPid].group ← group
7:      else
8:          GLP[LPid].group ← LPid
9:      end if
10:     if GLP[LPid].MaxDep ≠⊥ then
11:         GLP[LPid].group = REGROUP(GLP, GLP[LPid].MaxDep, GLP[LPid].group)
12:     end if
13:     return GLP[LPid].group
14: end procedure
```

LPs to worker threads. We rely on the *LpDependencies* matrix, which gets incremented at elements $[i, j]$ and $[j, i]$ whenever a cross-state access between LP_i and LP_j is detected. We map *LpDependencies* to an incidence matrix of a directed multigraph $G = (V, E)$ where the set of vertices V keeps the identifiers of the LPs in the system, and the set of edges E is defined as $E = \{\{i, j\} : i, j \in V \wedge LpDependencies[i, j] > 0\}$. Before converting it to an incidence matrix, we filter the values to reduce the possibility of capturing spurious cross-state relations, by using a threshold τ_{dep}. We thus build a *cross-state dependency multigraph* $G = \{\{i, j\} : i, j \in V \wedge LpDependencies[i, j] \geq \tau_{dep}\}$ and derive its incidence matrix IMG. If no edge exists in G between two LPs LP_i and LP_j, then the (i, j) IMG element's value is set to the special value $\perp$. Periodically, IMG is accessed to identify the highest cross-state access counter:

$$MaxDep_k = \max_{i \in [0, numLPs-1], i \neq k} \{IMG[k, i]\} \tag{3}$$

where $\perp$ is assumed to be the lowest value in the domain where the maximum is searched. These indices are used to build a vector of tuples, each one structured as $\langle MaxDep_k, group \rangle$ $\forall k \in [0, numLPs - 1]$. Initially, the value *group* for all the elements is set to $\perp$, telling that LP_k has its highest dependency counter set to $MaxDep_k$ and belongs to the special group $\perp$ (no group).

This construction transforms the multigraph G into another oriented multigraph $\bar{G}$ such that $\bar{V} \equiv V$, but if $\{i, j\} \in \bar{V}$, then $\{i, k\} \notin \bar{V} \quad \forall k \neq j$. This means that every node $i \in \bar{V}$ has at most one edge connecting it to another node $j \in \bar{V}$, with $i \neq j$, and by construction $j = MaxDep_i$.

A graph visiting algorithm on $\bar{G}$ is then used to group LPs together. We iterate over all indices $k \in [0, numLPs - 1]$, and for each value k we execute the recursive function REGROUP(*LpGroup*, k, $\perp$) shown in Algorithm 1. Its goal is to determine whether the selected LP already belongs to a group or, in the negative case, either the target LP is aggregated into the passed group (line 6) or a new group is created (line 8). In the positive case, only the group the LP belongs to is returned (line 3). Both cases are associated with *tentative groups*, which could be later confirmed or discarded. If the LP was associated with a tentative group, a recursive call is issued to REGROUP() (line 11), selecting as

the target LP the $MaxDep$ one of the current LP, and passing the ID of the group which the current LP belongs to. The group ID of the current LP is then updated with the return value of this call, which is done to backwards propagate the creation of new groups or the agglomeration to existing ones (line 13).

Once the graph visiting algorithm is completed, we apply Policy 1, taking into account the groups of LPs rather than single LPs. We note that in the scenario where no dependencies at all are detected, Algorithm 1 creates $numLPs$ groups, each one keeping a single LP. In this case, Policy 2 boils down to Policy 1.

4.3 Policy 3—Implicit and Explicit Synchronization

To account for both implicit and explicit synchronization, we must optimize towards multiple variables. For each LP_i of the system, we rely on a set of counters, identifying the volume of implicit and explicit interactions. Particularly, each LP_i is associated with a tuple $\langle I_0, I_1, \ldots, I_{numLP-1}, E_0, E_1, \ldots E_{numLP-1}\rangle$ where each component I_j is the amount of implicit accesses from LP_i to LP_j—measured in terms of cross-state synchronizations. Each E_j is the amount of events scheduled from LP_i to LP_j. For the case $i = j$, we arbitrarily set the value I_i to the number of events executed by LP_i, under the assumption that the likelihood that one LP accesses its own state is very high. This decision prevents the introduction of any bias in the general algorithm which is used for load-sharing.

Each tuple $\langle I_0, I_1, \ldots, I_{numLP-1}, E_0, E_1, \ldots E_{numLP-1}\rangle$ can be regarded as a point in an n-dimensional space, referred to as the *LPs interaction space*. The third policy aims at identifying a set of *clusters* of LPs with high interdependence. Indeed, if two LPs have similar coordinates in the n-dimensional space, they are very likely to interact. In particular, we want to identify K clusters, where K is the number of active worker threads. To this end, we rely on a variant of the Lloyd's solution [33] to the problem of finding evenly-sized Voronoi regions in an Euclidean space. This variant, known as the *k-medoids clustering algorithm* [34], tries to partition the available $numLP - 1$ LPs into K different clusters trying to minimize the effect of outliers. Specifically, if we call $\mathbf{i}$ and $\mathbf{j}$ the n-dimensional vectors associated with the coordinates of LP_i and LP_j in the n-dimensional interaction space, we define the *distance* between the two LPs as the Manhattan distance $d(\mathbf{i}, \mathbf{j}) = \|\mathbf{i} - \mathbf{j}\| = \sum_{i=1}^{n} |i_i - j_i|$. This distance is used in the objective function of the algorithm, which is defined as:

$$D = \sum_{k=1}^{K} \sum_{i \in C_k} \sum_{j \in C_k} d_{i,j} \tag{4}$$

where C_k is the set of all LPs in cluster k. When the load-sharing resource allocation is recomputed, an initial LP is selected having the shortest distance to any other LP in the n-dimensional space—it is *approximately* in the center. Then, other $k - 1$ LPs are selected so that they decrease the value of D as much as possible. In a second phase, possible alternatives for the k objects are selected, by picking an unselected LP and trying to exchange it with one of the k

objects. The choice is kept if and only if it produces a decrease in the value of D. This step is repeated until no exchange can be found that lowers the objective function's value. We anyhow impose a maximum number of refinement steps, which can be tuned at compile time.

The selected k LPs define the centroids of the k Voronoi regions of the n-dimensional interaction space. The LPs belonging to each group can then be picked minimizing the distance $d(\mathbf{i}, \mathbf{j})$ with respect to the centroids.

5 Experimental Results

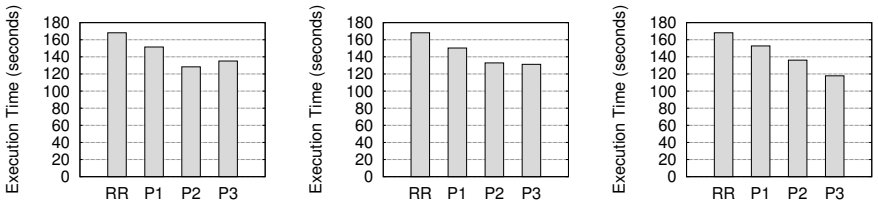
To study our policies, we rely on a synthetic benchmark which is representative of a wide range of agent-based models. Upon simulation startup, a pre-determined number of LPs acting as non-disjoint hexagonal cell regions is set up. They implement event handlers which, with a certain probability, operate changes on the hosted agents, execute an agent migration, or schedule to any other LP an *operative event*, i.e. an event associated with an operation correlating two agents hosted by different LPs.

As described, we map agents to data structures. An agent is described by a bitmask of attributes and a payload which is updated by the event handler implemented at any LP. In particular, we define three operations:

- *State-machine update*: with a certain probability p_{smu}, a bit in the bitmask is negated, mimicking a state transition;
- *Memory update*: with a certain probability p_{mu}, a portion of the payload of the agent's structure is written with random data, mimicking the update of less-concise metadata describing the agent;
- *Remote agent interaction*: with a certain probability p_{rai}, a random LP is scheduled an event piggybacking random data. Upon its receipt, a random agent is picked and the content of the event is copied into its state, mimicking kinship or family interactions with relatives who live in separate places.

Upon simulation startup, each LP instantiates the same number of agents, to have an even distribution, and links them to the primary list. Each LP schedules to itself separate chains of events, exponentially distributed, which trigger the state-machine and memory update operations. Once one of these operations is triggered, the LP scans the whole list of records so as to randomly select agents which undergo the corresponding operation. After a certain residence time, an agent is migrated towards one remote region, and a new agent migration event is scheduled, so that its lifetime within a certain region is pre-determined. Upon installation, with a certain probability p_{sh} the agent is shared (via message passing) with another region as well.

We have varied the probability p telling whether two LPs interact via message passing—$p = 0.5$ shows an even amount of in-place accesses vs message passing. We set $p_{smu} = 0.3$, $p_{mu} = 0.5$, $p_{rai} = 0.2$, and $p_{sh} = 0.1$, we use 1024 regions, with a population of 100.000 agents, and run the experiments on ROOT-Sim [16] on a 32-cores NUMA machine with 32 GB of RAM. The payload buffer of an



(a) $p = 0.25$ (b) $p = 0.50$ (c) $p = 0.75$

Fig. 2. Experimental results with different in-place state access probability p.

agent is 16KB, for a total of ∼1.6GB of live simulation state (i.e., without considering checkpoints). Additionally, we compare to an "agnostic" load sharing, where LPs are bound to threads in a round robin fashion (RR in the plots). By the results in Figure 2, we can see that when the amount of message-passing interactions is non-minimal (Figure 2(c)), Policy 3 offers the better results. In fact, this is the only policy which accounts for both implicit and explicit interaction among LPs. On the other hand, when the vast majority of the interactions rely on in-place accesses (Figure 2(a)), Policy 2 gives better results, although in a slightly reduced way since the graph visiting algorithm is not able to capture a large amount of mutual dependencies. Policy 1 is interaction-agnostic, and is not therefore able to compete with the other two policies.

In the best case, there is a performance speedup of around 30% with respect to the RR policy. This evidences that load-sharing policies are fundamental to offer a competitive simulation when run on shared-memory systems.

6 Conclusions

In this paper we have discussed a parallel ABM programming model for demography, using the DES formalism. Additionally, we have proposed three different policies to support efficient load balancing under different workloads. By our results, we showed how load balancing is fundamental when running simulations on shared-memory machines. Moreover, policies which explicitly account for (implicit and explicit) interactions can find a binding between LPs and threads which allows to better capture the parallelism degree of the model, and thus increase performance.

References

1. Jennings, N.R., Sycara, K., Wooldridge, M.: A roadmap of agent research and development. Autonomous agents and multi-agent systems **1**(1) (1998) 7–38
2. Takahashi, T., Tadokoro, S., Ohta, M., Ito, N.: Agent based approach in disaster rescue simulation-From test-bed of multiagent system to practical application. In: Robot Soccer World Cup V. RoboCup, Springer-Verlag (2002) 102–111

3. Balbo, A.L., Rubio-Campillo, X., Rondelli, B., Ramírex, M., Lancelotti, C., Torrano, A., Salpeteur, M., Lipovetzky, N., Reyes-García, V., Montañola-Sales, C., Madella, M.: Agent-based simulation of Holocene monsoon precipitation patterns and hunter-gatherer population dynamics in semi-arid environments. Journal of Archaeological Method and Theory **21**(2) (2014) 426–446
4. Prats, C., Montañola-Sales, C., Gilabert-Navarro, J.F., Valls, J., Casanovas-Garcia, J., Vilaplana, C., Cardona, P.J., López, D.: Individual-based modeling of tuberculosis in a user-friendly interface: Understanding the epidemiological role of population heterogeneity in a city. Frontiers in Microbiology **6**(1564) (jan 2016)
5. Page, S.E.: Agent-based models. In Durlauf, S.N., Blume, L.E., eds.: The New Palgrave Dictionary of Economics. Nature Publishing Group (2008) 47–52
6. Tisue, S., Wilensky, U.: Netlogo: A simple environment for modeling complexity. In: Proceedings of the International Conference on Complex Systems. ICCS, NECSI (2004) 1–10
7. Minar, N., Burkhart, R., Langton, C., Askenazi, M.: The SWARM simulation system: A toolkit for building multi-agent simulations. Technical report, Santa Fe Institute (1996)
8. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: MASON: A multi-agent simulation environment. Simulation **81**(7) (2005) 517–527
9. North, M.J., Howe, T.R., Collier, N.T., Vos, J.R., M.J. North, T.R. Howe, N.T. Collier, J.V.: The Repast simphony runtime system. In: Proceedings of the Agent 2005 Conference on Generative Social Processes, Models and Mechanisms, Argonne National Laboratory (2005) 151–158
10. Lysenko, M., D'Souza, R.M.: A framework for megascale agent based model simulations on the GPU. Journal of Artificial Societies and Social Simulation **11**(4) (2008) 10
11. Park, H., Han, J.: Fast rendering of large crowds using GPU. In: Entertainment Computing. Springer Berlin Heidelberg (2008) 197–202
12. Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., Spagnuolo, C.: A framework for distributing agent-based simulations. In Alexander, M., D'Ambra, P., Belloum, A., Bosilca, G., Cannataro, M., Danelutto, M., Martino, B., Gerndt, M., Jeannot, E., Namyst, R., Roman, J., Scott, S.L., Traff, J.L., Vallée, G., Weidendorfer, J., eds.: Proceedings of Euro-Par 2011: Parallel Processing Workshops. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2012) 460–470
13. Fujimoto, R.M.: Parallel discrete event simulation. Communications of the ACM **33**(10) (1990) 30–53
14. Jefferson, D.R.: Virtual Time. ACM Transactions on Programming Languages and System **7**(3) (1985) 404–425
15. Barnes, P.D., Carothers, C.D., Jefferson, D.R., LaPre, J.M.: Warp speed: executing time warp on 1,966,080 cores. In: Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation - SIGSIM-PADS '13. (2013) 327
16. Vitali, R., Pellegrini, A., Quaglia, F.: Towards symmetric multi-threaded optimistic simulation kernels. In: Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation. PADS, IEEE Computer Society (jul 2012) 211–220
17. Pellegrini, A., Quaglia, F.: Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies. In: Proceedings of the 2014 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation. PADS, ACM Press (2014) 105–116
18. Pellegrini, A., Quaglia, F.: Programmability and performance of parallel ECS-based simulation of multi-agent exploration models. In: Proceedings of Euro-Par 2014: Parallel Processing Wor, Porto, Portugal, LNCS, Springer-Verlag (2014)

19. Carothers, C.D., Fujimoto, R.M.: Efficient execution of Time Warp programs on heterogeneous, NOW platforms. IEEE Transactions on Parallel and Distributed Systems **11**(3) (2000) 299–317
20. Glazer, D.W., Tropper, C.: On process migration and load balancing in Time Warp. IEEE Transactions on Parallel and Distributed Systems **4**(3) (1993) 318–327
21. Vitali, R., Pellegrini, A., Quaglia, F.: Load sharing for optimistic parallel simulations on multi core machines. ACM SIGMETRICS Performance Evaluation Review **40**(3) (jan 2012) 2–11
22. Karpov, Y.G.: AnyLogic — a New Generation Professional Simulation Tool. In: Proceedings of the 6th International Congress on Mathematical Modeling. MATHMOD (2004)
23. Holcombe, M., Coakley, S., Smallwood, R.: A general framework for agent-based modelling of complex systems. In: Proceedings of the 2006 European conference on complex systems, European Complex Systems Society Paris, France (2006)
24. Richmond, P., Romano, D.: Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu. In: Proceedings International Workshop on Supervisualisation. (2008)
25. Vitali, R., Pellegrini, A., Quaglia, F.: Assessing load sharing within optimistic simulation platforms. In: Proceedings of the 2012 Winter Simulation Conference. WSC, Society for Computer Simulation (2012)
26. Vitali, R., Pellegrini, A., Quaglia, F.: A Load Sharing Architecture for Optimistic Simulations on Multi-Core Machines. In: Proceedings of the 19th International Conference on High Performance Computing. HiPC, IEEE Computer Society (2012) 1–10
27. Marziale, N., Nobilia, F., Pellegrini, A., Quaglia, F.: Granular Time Warp objects. In: Proceedings of the 2016 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation. PADS, New York, New York, USA, ACM Press (2016) 57–68
28. Bauer, P., Lindén, J., Engblom, S., Jonsson, B.: Efficient Inter-Process Synchronization for Parallel Discrete Event Simulation on Multicores. In: Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation - SIGSIM-PADS '15, New York, New York, USA, ACM Press (2015) 183–194
29. Andrew, H.: Demographic Methods. Routledge (1998)
30. Montañola-Sales, C., Casanovas-Garcia, J., Kaplan-Marcusán, A., Cela-Espín, J.M.: Demographic agent-based simulation of Gambians immigrants in Spain. In: Proceedings of the 10th Social Simulation Conference, European Social Simulation Association (2014)
31. Cingolani, D., Pellegrini, A., Quaglia, F.: RAMSES: Reversibility-based agent modeling and simulation environment with speculation support. In Hunold, S., Costan, A., Ginenéz, D., Iosup, A., Ricci, L., Gómez Requena, M.E., Scarano, V., Varbanescu, A.L., Scott, S.L., Lankes, S., Weidendorfer, J., Alexander, M., eds.: Proceedings of Euro-Par 2015: Parallel Processing Workshops. PADABS. LNCS, Springer-Verlag (2015) 466–478
32. Dantzig, G.B.: Discrete-variable extremum problems. Operational Research (5) (1957)
33. Lloyd, S.: Least squares quantization in PCM. IEEE Transactions on Information Theory **28**(2) (mar 1982) 129–137
34. Kaufman, L., Rousseeuw, P.J.: Clustering by means of medoids. Statistical Data Analysis Based on the L1-Norm and Related Methods (1987) 405–416416