An Adaptive Load Management Mechanism for Distributed Simulation of Multi-agent Systems

Ton Oguara, Dan Chen and Georgios Theodoropoulos
School of Computer Science, University of Birmingham,
Birmingham B15 2TT, UK
Email: {txo|cxd|gkt}@cs.bham.ac.uk

Brian Logan and Michael Lees
School of Computer Science and Information Technology
University of Nottingham, Nottingham NG8 1BB, UK
Email:{bsl|mhl}@cs.nott.ac.uk

Abstract

The paper presents a load management mechanism for distributed simulations of multi-agent systems. The mechanism minimises the cost of accessing the shared state in the distributed simulation by dynamically redistributing shared state variables according to the access pattern of the simulation model. To evaluate the effectiveness and performance of the mechanism, a series of benchmark experiments were performed using the PDES-MAS framework for distributed simulation of multi-agent systems. Although preliminary, the results indicate that the proposed mechanism significantly reduces the overall access cost of the system.

1 Introduction

Multi-agent-based systems (MAS) are increasingly being applied in a wide range of areas, including telecommunications, business process modelling, software system design and computer games etc. They are particularly appropriate in large, complex or unpredictable domains, where multiple agents interact and collaborate to solve the problem [18].

Simulation has traditionally played an important role in multi-agent system research and development as it allows a degree of control over experimental conditions and facilitates the replication of results in a way that is difficult or impossible with a prototype or fielded system, freeing the agent designer or researcher to focus on key aspects of a system.

With multi-agent systems becoming ever more larger and complex, distributed simulation has attracted the interest of the MAS community research [1, 15]. In [9, 11] distributed simulation technology has been identified as an effective and flexible approach for designers to investigate MAS. Decentralized, event-driven distributed simulation is particularly suitable for modelling with inherent asynchronous parallelism, such as MAS. However, although it is relatively straightforward to model agents using conventional parallel and distributed simulation techniques, it is harder to model and simulate agents' environment.

In conventional parallel discrete event simulation, the model is partitioned into a fixed number of Logical Processes, each of which maintains its own portion of the state of the simulation. The interaction between the processes is known in advance and does not change during the simulation. In contrast, simulations of MAS typically have a large *shared* state, the agents' environment, which is only loosely associated with any particular process. During execution, each individual process may access different parts of the shared state at different times and the pattern of access is nondeterministic and therefore difficult to predict. The efficient simulation of the agents' environment is a key problem in the distributed simulation of MAS as the management of shared state can have a significant impact on the overall system performance. Managing the state (or, in other words, modelling the agents' environment) in a single centralised process can be a bottleneck, while distributing the state among all different LPs will result in frequent broadcasting (a problem also relating to Interest Management).

In an effort to address this problem and provide a generic approach for dynamic load balancing and interest management in distributed simulation of MAS, in [11, 19] we in-

troduced the notion of *Spheres of Influence (SoI)* as an approach to dynamically decompose and distribute the shared state. The *SoI* describe the access pattern of the simulation models on the shared state in a given time interval. In more recent work we have realised this approach in the PDES-MAS framework [11].

PDES-MAS adopts a standard discrete event simulation approach with optimistic synchronization, while the shared state is maintained by a tree of logical processes (the *Communication Logical Processes (CLP)*, which clusters agent models and shared state according to the agents' *SoI*. In [8] we have represented the shared state variables (SSVs) of the simulation as a distributed set of tuple spaces, while in [9] we have provided a more detailed description of the CLPs, their internal architecture and their operation and in [10] we have discussed synchronisation issues.

In this paper we describe an adaptive load management mechanism used by PDES-MAS to approximate the ideal *SoI* of the overall simulation. The load management mechanism exploits a migration algorithm to move shared state variables in accordance with the *SoI*, redistributing the shared state to best adapt to the access pattern of the simulation LPs.

The remainder of this paper is organized as follows. In section 2 we introduce the PDES-MAS framework and briefly describe the internal design of the simulation models and the hierarchical infrastructure of CLPs. The shared state variable migration algorithm is outlined in section 3. In section 4 we present preliminary results of experiments to investigate the performance of the migration algorithm. In section 5 we discuss related work and in section 6, we conclude with a summary and discussion of future work.

2 The PDES-MAS framework

In the context of PDES-MAS framework, a multi-agent system is modelled as a network of *Logical Processes (LPs)*. In particular, each agent is modelled as an *Agent Logical Process (ALP)* [9].

ALPs have both private state and shared state. The private state is maintained within the ALP, while the shared state can be accessed (read or written) by other LPs. The shared state is modelled as a set of *Shared State Variables (SSVs)*. Any operation on the shared state by an ALP is called an external event, which is modelled as the exchange of a timestamped message between the LPs.

Different types of external event will have different effects on the shared state. The sphere of influence of an event is defined as the set of shared state variables read or written by the event [11]. The sphere of influences of an LP p_i over time interval $([t_1, t_2])$, $s(p_i)$ is the union of spheres of influence of events generated by p_i. Intersecting the spheres of influence for each event generated by the LP gives a partial order over sets of state variables for the LP over the interval, which denotes the frequency with which those SSVs been accessed.

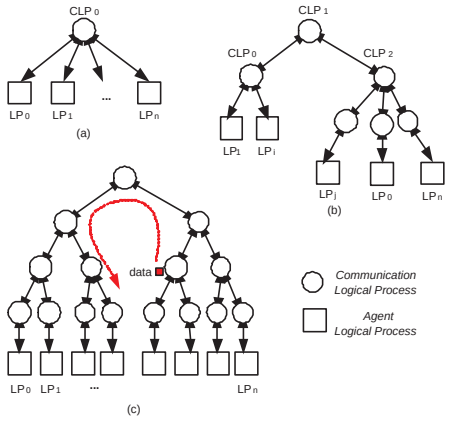


Figure 1. Illustrating the PDES-MAS Framework

In addition to ALPs, the PDES-MAS framework contains *Communication Logical Processes (CLPs)*, which manage the shared state. In [11], we propose that the simulation initially starts with a single centralised CLP marked as CLP_0 (see figure 1). As the simulation execution proceeds, CLP_0 may become a bottleneck due to the increasing load of managing SSVs. When this happens, CLP_0 is split into multiple CLPs, with each new CLP maintaining a subset of the shared state which is most closely associated (in terms of their spheres of influence) with the ALPs which are below it in the tree. This reduces the load on individual CLPs, and such "decomposition" of congested CLPs naturally leads to the construction of a CLP tree (figure 1(b)). As the access patterns on the shared state change, so does the configuration of the tree and the distribution of state (i.e., its allocation to CLPs) to reflect the logical topology of the model. Redistribution can be achieved in a number of ways, such as: 1.) Moving CLPs, 2.) Moving ALPs and 3.) Moving state between CLPs. In this paper we choose to use a fixed tree of CLPs and move SSVs through the tree to achieve redistribution.

Figure 2 gives a schematic view of a typical CLP. CLPs interact with other LPs via ports. Ports link the individual LPs together to form the overall PDES-MAS simulation system. The CLP tree works as a runtime infrastructure providing common services to the ALPs. The services provided by the CLP tree include: (1) facilitating the fabric of the distributed simulation; (2) clustering and interoperating the ALPs; (3) managing shared state and balancing load incurred by accessing the shared state; and (4) facilitating synchronization of the ALPs. Operation of the CLP

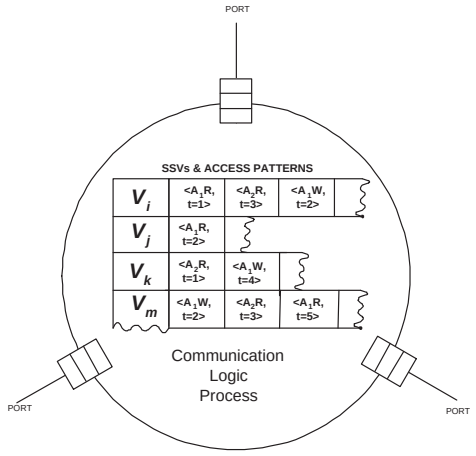


Figure 2. CLP and Ports

tree remains transparent to the ALPs during the simulation. Figure 3 illustrates the relationship between a ALP and the CLP tree.

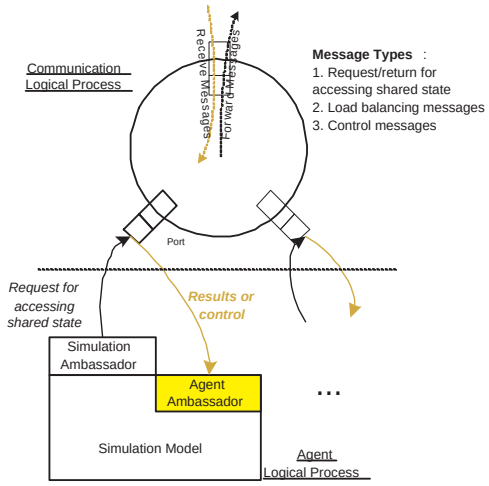


Figure 3. Relationship between the CLP Tree and Agent/Environment LPs

An ALP issues requests to access shared state variables through a *SimulationAmassador* module which forwards the requests to the port of corresponding leaf CLP. If the required SSV is not held locally, the CLP forwards the request to the appropriate CLP. Return value and control messages (e.g. rollback) are conveyed to the ALP via its *AgentAmbassador* module.

3 Shared State Variable Migration Algorithm

In a distributed simulation based on the PDES-MAS framework, access requests to SSVs are initiated by ALPs and are routed through the tree and processed by the CLPs. The history of accesses to each SSV is recorded by the CLPs. The SSV migration algorithm evaluates the cost of these accesses to determine when and how to migrate SSVs. As some access patterns may cause the concentration of a significant proportion of he shared state in few CLPs and introduce new bottlenecks, the migration algorithm also takes the load distribution of CLPs into account in making decisions. Hence, the mechanism minimises the cost of accessing shared state while balancing the load of managing shared state.

In the context of PDES-MAS framework, we define the **rank** of an SSV v_j for LP p_i, over the time interval $([t_1, t_2])$, $r(v_j, p_i)$ as the number of events in whose sphere of influence v_j lies (ie., the number of access from p_i to v_j). The **distance** from state variable v_j to the ALP p_i, $l(v_j, p_i)$ is defined as the number of hops from p_i to the CLP maintaining v_j. We can now define the cost of accessing a state variable v_j from an ALP p_i, as

$$c(v_j, p_i) = r(v_j, p_i) \times l(v_j, p_i)$$

then the total cost of accessing a state variable v_j from a port P_k, as

$$c(v_j, P_k) = \sum_{\forall p_i beyond P_k} c(v_j, p_i)$$

The total cost of all access to v_j is then

$$c(v_j) = \sum_{\forall p_i} c(v_j, p_i)$$

The total access cost at CLP_l is then,

$$c(CLP_l) = \sum_{\forall v_j \in CLP_l} \sum_{\forall P_k \in CLP_l} c(v_j, P_k)$$

Finally the overall access cost of the whole tree is defined as,

$$c_{tree} = \sum_{\forall CLP_l} c(CLP_l)$$

The optimal decomposition of SSVs over this time interval is the one which minimizes the total access cost of the tree, (c_{tree}). To adapt to changes in the spheres of influence, the SSVs in the CLP tree must be redistributed periodically. Clearly, the closer an SSV is located to the ALPs which access it most frequently, the smaller the total access cost is.

For the purposes of migration, the computational load on a CLP is taken to be a function of the number of messages

processed by the CLP. More precisely, the computational load of a CLP_j, L_{CLP_j}, is defined as the total the number of accesses to the SSVs maintained at the CLP.

Only when the computational load exceeds a predefined threshold (denoted by variable $T_{H_{load}}$), will the CLP consider whether to invoke SSV migration. The algorithm then uses the total access cost of each SSV ($c(v_j)$) to determine which SSVs should be migrated, any SSV where $c(v_j) > T_{H_{var}}$. The next stage involves determining which port each selected SSV should be pushed through. A port P_k is selected if $c(v_j, P_k)$ is greater than the summation of the access cost on v_j through all other ports plus the threshold $T_{H_{port}}$. Another task of the SSV migration algorithm is to ensure an even distribution of the computational load among CLPs. This is achieved by preventing a CLP from becoming overloaded due to accepting too many SSVs (the L_{PUSH_j} load) from its neighbours. $T_{H_{swap}}$ denotes the maximum difference in load allowed between neighbouring CLPs.

Figure 4 illustrates the SSV migration algorithm from the point of view of both the CLP which initiates the SSV migration (the "sender") and its neighbour CLP that accepts the migrated SSV (the "receiver"). The SSV migration algorithm can be described as follows:

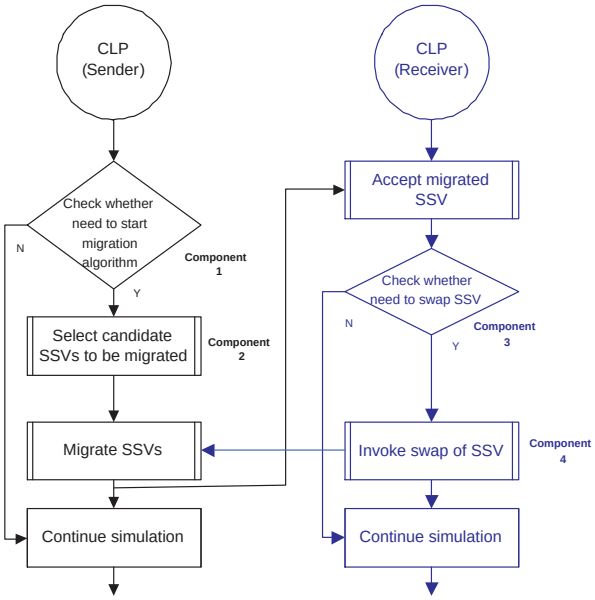


Figure 4. Illustrating the Shared State Variable Migration Algorithm

- Collecting information: The CLPs periodically collect load information in order to determine whether to initiate SSV migration. For instance, CLP_j computes L_{CLP_j} and compares L_{CLP_j} with $T_{H_{load}}$ before initiating SSV migration. When the migration algorithm is invoked, it computes $c(v_j)$ for each local SSV, the access cost $c(v_j, P_k)$ for each SSV, and the access cost for CLP_j. Based on this information, the decision regarding which shared state variables should be pushed and where they should be pushed can be made.

- Sorting candidate SSVs: The algorithm then sorts the cost values for each SSV in descending order and checks each cost against the preset cost threshold $T_{H_{var}}$. All variables which have an access cost over the last period greater than $T_{H_{var}}$ will be selected as candidates for migration.

- Determining SSVs to be migrated: Once the set of migration candidate SSVs has been determined, the CLP chooses to which of the neighbouring CLPs each candidate should be pushed (if any). For each candidate SSV, the algorithm determines the port through which the cost of accessing the SSV $c(v_j, P_k)$ is highest. If this access cost is greater than $T_{H_{port}}$ the variable is considered for migration through this port. $T_{H_{port}}$ is specified proportional to the total access cost for the CLP, thus it adapts to the possible fluctuations of cost at runtime. For example, for candidate SSV v_j, suppose $c(v_j, P_k)$ is the largest, if $c(v_j, P_k)$ is greater than $\sum_m \{c(v_j, P_m) | m \neq k\} + T_{H_{port}}$, then v_j will be "provisionally" selected for migration.

- Determining destination port: Once an SSV is chosen for migration, the algorithm needs to decide which port to push that SSV through. The access cost of the SSV at its potential destination CLP is computed based on the assumption that the pattern of accesses in the near future is similar to the last time interval [11]. If the "new" access cost is lower than present, the SSV will be pushed to the new location. Thus, the algorithm therefore ensures that the overall communication cost will be reduced after migration. This procedure helps to avoid thrashing caused by the "ping-pong" of the same SSV between two neighbouring CLPs.

- Swapping SSVs: Obviously, the CLP accepting a migrated SSV incurs additional computational load. This CLP may need to "swap" some load with the sender if the receiving CLP itself has a high computational load. The algorithm checks this condition by computing the difference between the computational loads of the receiver and sender. The receiver will choose an SSV to swap only if two conditions are met:

  1. the swap will eventually reduce the access cost of the SSV; and

2. the majority of its accesses are through the port through which the sender pushed the new load.

At the original sender side, the CLP checks incoming messages from the original receiver. If it is a swap message, the CLP needs to process it and avoid thrashing.

Figures 5,6,7,8 give the pseudo-code for the four major functional components in the migration algorithm. Components 1 and 2 describe the behavior of the sender CLP whereas components 3 and 4 are invoked by the receiver CLP. Component 1 checks whether the CLP should invoke the migration procedure. Component 2 selects the candidate SSVs to be migrated. Once the SSVs are pushed to the receiver, the receiver invokes component 3 to determine whether it is necessary to swap some SSVs with the sender. If necessary, the receiver will use component 4 to choose SSVs to push to the sender before it continues with normal execution.

```
/* Check to know if state migration is necessary.
 * Executed at State Migration initiating node - CLP_j
 */
if ( L_CLP_j > T_H_load ) then
    select a SSV or a set of SSVs to migrate;
    specify the port through which the SSV will be pushed;
    migrate a SSV or set of SSVs from node CLP_j to CLP_m;
```

Figure 5. Start of SSV Migration (Component 1)

```
/* select SSV/SSVs to migrate. Executed at SSV migration initiating node - CLP_j */
for each v_j ∈ CLP_j do
    compute c(v_j); /* total cost of accessing v_j on CLP_j */
    if ( c(v_j) > T_H_cost ) then
        for each P_k ∈ CLP_j then /* identify port with highest access cost to v_j */
            compute c(v_j, P_k); /* total cost of accessing v_j via port P_k on CLP_j. x ≠ k */
            if ( c(v_j, P_k) > ( Σ_x c(v_j, P_x) + T_H_concurr ) ) then
                compute cf(v_j); /* possible future cost to v_j, if v_j is moved to CLP_m */
                if ( cf(v_j) < c(v_j) ) then /* confirm gain in pushing v_j to CLP_m */
                    P[L] ← v_j; /* add v_j to the push list P[L] */
                    set the push port for v_j as P_k;
/* migrate a SSV/ SSVs from node CLP_j via port P_k to CLP_m */
for each v_j ∈ P[L] do
    L_PUSH_j ← L_PUSH_j + a(v_j); /* a(v_j) is the total accesses to variable v_j */
    PUSH ( P[L], L_CLP_j, L_PUSH_j ); /* push a SSV/ SSVs from CLP_j to CLP_m */
```

Figure 6. Selection of Shared State Variable for Migration (Component 2)

```
/* State variable acceptance. Executed at the node accepting state variable - CLP_m.
* CLP_m receives the state variable(s) from CLP_j via port P_x on CLP_m
*/
if (msg_tag == load_tag) then
    ACCEPT ( P[L], L_CLP_j, L_PUSH_j ); /* accept the push list from CLP_j */
    compute L_CLP_m; /* L_CLP_m is the computational load of CLP_m */
    /* load variance L_v between CLP_j and CLP_m */
    L_v = | (L_CLP_m - L_PUSH_j) - (L_CLP_j - L_PUSH_j) |;
    if ( L_v > T_H_swap ) then
        compute SWAP_load using the SWAP algorithm;
```

Figure 7. Acceptance of Shared State Variable (Component 3)

4 Experimental Results

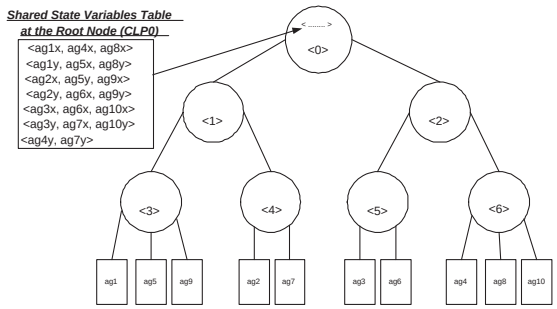
To evaluate the load management algorithm described in the previous section, we have run a number of experiments using traces generated from SIM_BOIDS, an implementation of Boids [14]using the SIM_AGENT [17] toolkit. Boids is a simulation of a coordinated animal motion such as bird flocks or fish schools. In SIM_BOIDS, the environment consists solely of the agents x and y position.

Figure 9 shows the experimental scenario with a tree of seven CLPs and ten Boids agents (ALPs) (ag1,..., ag10). The shared state consists of twenty state variables (the x and y positions of the ten agents) which are initially located at the root CLP node (figure 9(a)). The SSVs are then gradually moved down the tree (figure 9(b)) by the SSV migration algorithm. The ALPs are randomly assigned at the leaf nodes. All reported results are the average from 5 executions.

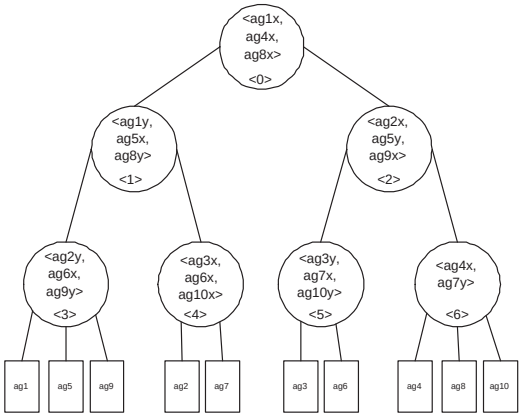
Figure 10 shows results with state migration (SM) and no state migration(NoSM), where the initial configuration

```
/* Compute SWAP_load using the SWAP algorithm. SSV(s) are received via Port P_x.
* Executed at the node accepting SSV(s) - CLP_m */
sort all SSVs maintained at CLP_m and accessed via Port P_x, using c(v_m) as sort keys;
SWAP_load ← 0;
for each v_m in sort_list do
    compute c(v_m, P_x); /* total cost of accessing v_m via port P_x on CLP_m */
    SWAP_load ← SWAP_load + a(v_m); /* a(v_m) is the total accesses to variable v_m */
    if ( (SWAP_load - L_PUSH_j) > T_H_swap ) then
        compute cf(v_m); /* possible future cost to v_m, if v_m is swapped with CLP_j */
        if ( cf(v_m) < c(v_m) ) then /* confirm gain in swapping v_m with CLP_m */
            S[L] ← v_j; /* add v_m to the swap list S[L] */
            set the push port for v_m as P_x;
    if ( (SWAP_load - L_PUSH_j) ≤ T_H_swap ) then
        return NIL
/* SWAP SSV/ SSVs with node CLP_j via port P_x */
for each v_m ∈ S[L] do
    SWAP ( S[L]); /* SWAP SSV/SSVs with CLP_j */
```

Figure 8. Swap of Shared State Variable (Component 4)



(a) Initial distribution of SSVs (at the root CLP)



(b) Redistribution of SSVs after migration

Figure 9. Structure of the Experiment Scenario

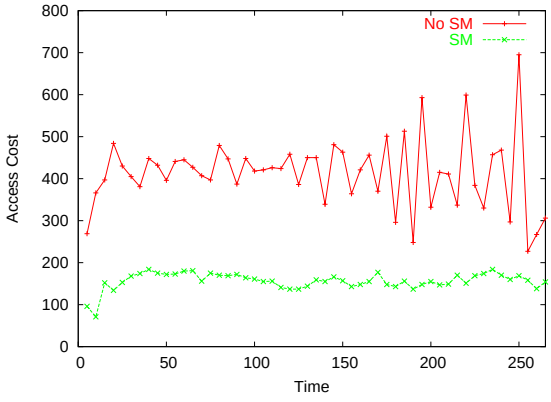


Figure 10. Access cost comparison between using and not using SSV migration

for both is with all SSVs at the root CLP node. By assigning all SSVs to the root node, we achieve an average cost of accessing each SSV (as the distance is the same for all agents)[1]. For the experiment, $T_{H_{load}}$=0, $T_{H_{var}}$=0, $T_{H_{port}}$=25% and $T_{H_{swap}}$=2. Snapshots of access costs for each CLP are obtained periodically in real time, while the results presented are for the first 100 clock cycles of the simulation. Figure 10 shows that with state migration, the overall access cost of the simulation is reduced from approximately 400 with no state migration to 156 with state migration, a reduction of 61%. The figure also shows fluctuations in access cost with no state migration as a result of agents changing their patterns of access. These fluctuations are reduced by state migration, with the overall access cost remaining largely constant for the entire simulation.

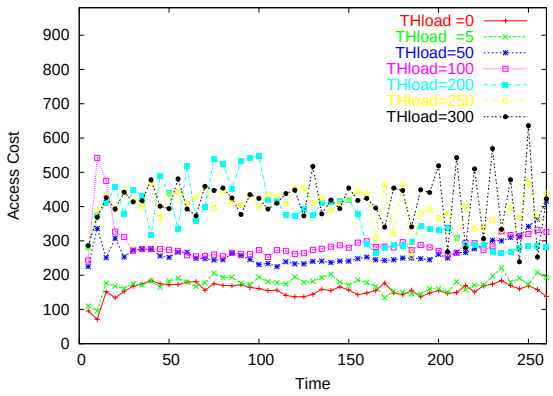


Figure 11. Impact on Access Cost using Different Thresholds

Figure 11 illustrates the impact of the frequency of state migrations to the total access cost. In this particular experiment, the frequency is determined by the value of the $T_{H_{load}}$ threshold. For small threshold values, the tree reflects more accurately the changes in the *SoIs* in the system, as SSVs migrate sooner to their appropriate position in the tree. As a result, the smaller the threshold value, the lower the total access cost for the entire tree throughout the simulation. The access cost with $T_{H_{load}}$=0 is the smallest (about 156) while the access costs with $T_{H_{load}}$=250 and $T_{H_{load}}$=300 are the largest (about 400, in which case the variables will hardly ever migrate from the root node).

However, with frequent migrations the overall overhead for migrating SSVs increases. Figure 12 gives an indication of the potential overhead incurred for more frequent migrations of SSVs, in terms of the number of protocol messages for pushing/swapping SSVs between CLPs. Clearly, the

[1]With a random assignment of SSVs to CLPs the distance for each SSV may lie anywhere between 1-5

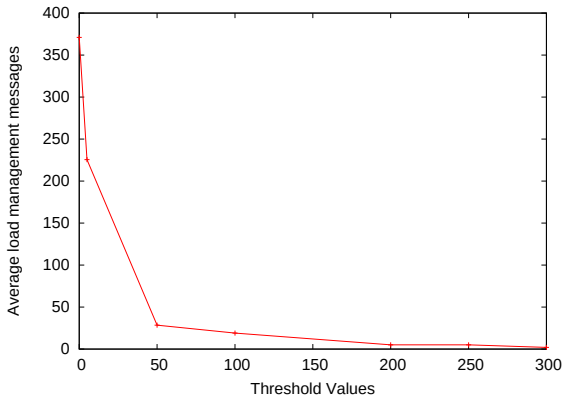


Figure 12. Messages for Migrating SSVs with Different Access Cost Thresholds

number of messages drops rapidly with the threshold less than 100 (from 371 to 19). Figures 11 and 12 suggest that the total access costs are very close with $T_{H_{load}}$=0 and 5 while the load management messages reduced dramatically (by 39%).

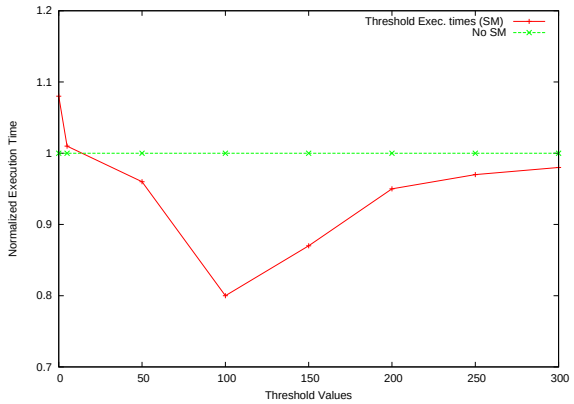


Figure 13. Performance Comparison of various threshold values

In figure 13, we compare the performance of the state migration algorithm at various threshold values in terms of execution time. The results are normalised with respect to the execution time when no state migration is applied (i.e. NoSM). It can be observed that at very low $T_{H_{load}}$ values, there is no significant improvement in the execution time of the algorithm when compared to the NoSM scenario. However as $T_{H_{load}}$ increases further, the execution time drops. The figure shows that the execution time of the algorithm improved by 20% with $T_{H_{load}}$=100. Notwithstanding, at $T_{H_{load}}$ values beyond 100, the overall execution time of the algorithm begin to rise again.

Obviously from figure 11, we can observe that with increasing $T_{H_{load}}$ values, the access cost increases while figure 12 shows a fall in the migration overhead. Thus at some stage the migration overhead and the reduction of access cost will balance. This is likely to result in the minimization of the overall execution time.

5 Related Work

There is a considerable amount of work in the simulation literature on the efficient distribution of updates, particularly in the context of large-scale real-time simulations where it is termed Interest Management (or Data Distribution Management). These techniques utilise filtering mechanisms to provide simulation models with only that subset of information relevant to them. In most existing systems, Interest Management is realised via the use of IP multicast addressing. Typically, the definition of the multicast groups of receivers is static, based on a priori knowledge of communication patterns between the simulation models. Indeed, most real time large scale distributed simulators such as CCTT [12], STOW ED-1 [3] and the High Level Architecture (HLA)-based DMSO RTI [5] utilise static interest management schemes. However, unlike the state migration algorithm presented above, static, grid-based Interest Management schemes do not adapt to dynamic changes in the communication patterns between simulation models during execution and are therefore incapable of balancing the communication and computation load. Although there have been a few attempts to define dynamic schemes for Interest Management which concentrate on the dynamic configuration of multicast groups within the context of HLA [2, 13, 20], the problem of dynamic interest management remains largely unsolved.

Another objective of PDES-MAS framework is to evenly distribute the communication and computational load of accessing and managing shared state. Load balancing has been studied extensively in both conservative and optimistic parallel simulation e.g. [4, 7, 16]. However, the issue of dynamic load balancing has received very little attention in relation to interest management and work in this area to date is either preliminary [21] or specific to particular applications [6].

6 Summary and Future Work

This paper presented an adaptive load management mechanism for the distributed simulation of multi-agent systems. The mechanism is applied in the PDES-MAS simulation framework. The core module of the mechanism is

a shared state variable (SSV) migration algorithm designed based on the notion of spheres of influence (*SoI*). The SSV migration algorithm dynamically distributes the SSVs to move them as close as possible to the ALPs which access them the most frequently. Thus, the ideal *SoI* can be approximated, minimising the overall cost of accessing SSVs. The experimental results indicate that the SSV migration algorithm can significantly reduce the cost of accessing SSVs. The results also prove that the algorithm considerably improves the execution efficiency of the overall system.

An MPI-based PDES-MAS simulation kernel has been developed and currently the synchronisation, load management and routing components are being integrated. This will enable further experimentation to evaluate the algorithm while also considering other factors (such as processor utilisation, impact on synchronisation etc). In addition, we intend to apply and evaluate the performance of the algorithm in other non agent related simulation environments (in particular mobile ad-hoc networks).

Acknowledgment

The authors would like to thank Dr. Yi Zhang at the Midlands e-Science Center (MeSC) for his support and for facilitating access to the MeSC cluster. This work is part of the PDES-MAS project[2] and is supported by EPSRC research grant No. GR/R45338/01.

References

[1] J. Anderson. A generic distributed simulation system for intelligent agent design and evaluation. In *Proceedings of the 10th International Conference on AI, Simulation, and Planning in High Autonomy Systems*, pages 36–44, March 2000.

[2] A. Berrached, M. Beheshti, O. Sirisaengtaksin, and A. Korvin. Alternative approaches to multicast group allocation in hla data distribution. In *Proc. Of the 1998 Spring Simulation Interoperability Workshop*, 1998.

[3] J. Calvin, C. Chiang, and D. V. Hook. Data subscription. In *12th Workshop on Standards for the Interoperability of Distributed Simulations*, pages 807–813, March 1995.

[4] C. Carothers and R. Fujimoto. Background execution of time-warp programs. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, pages 12–19. IEEE Computer Society Press, May 1996.

[5] DMSO. *High Level Architecture Run-Time Infrastructure - RTI 1.3-Next Generation Programmer's Guide Version 5*. Department of Defense(USA), Defense Modeling and Simulation Office, Feb 2002.

[6] C. Georgousopoulos and O. Rana. Combining state and model-based approaches for mobile agent load balancing. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 878–885. ACM Press, 2003.

[7] D. Glazer and C. Tropper. On process migration and load balancing in time-warp. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4, pages 318–327. IEEE Press, March 1993.

[8] M. Lees, B. Logan, R. Minson, T. Oguara, and G. Theodoropoulos. Distributed simulation of MAS. In *Multi-Agent and Multi-Agent-Based Simulation, Joint Workshop (MABS) 2004*, volume 3415 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2004.

[9] M. Lees, B. Logan, R.Minson, T. Oguara, and G. Theodoropoulos. Modelling environments for distributed simulation. In *First International workshop on Environments for Multi-Agent Systems (E4MAS 2004), in conjunction with the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS04)*, volume 3374 of *Lecture Notes in Computer Science*, pages 150–167. Springer, July 2004.

[10] M. Lees, B. Logan, and G. Theodoropoulos. Time windows in multi-agent distributed simulation. In *Proceedings of the 5th EUROSIM Congress on Modelling and Simulation (EuroSim'04)*, Sep. 2004. (to appear).

[11] B. Logan and G. Theodoropoulos. The distributed simulation of multi-agent systems. In *Proceedings of the IEEE*, volume 89, pages 174–186, 2001.

[12] T. Mastaglio and R. Callahan. A large-scale complex virtual environment for team training. *IEEE Computer*, 28(7):174–186, July 1995.

[13] K. Morse, L. Bic, M. Dillencourt, and K. Tsai. Multicast grouping for dynamic data distribution management. In *Proceedings of the 31st Society for Computer Simulation Conference*, pages 312–318, July 1999.

[14] C. Reynolds. Boids. Online: `http://www.red3d.com/cwr/boids`, 2005.

[15] B. Schattenberg and A. Uhrmacher. Planning agents in JAMES. *Proceedings of the IEEE*, 89(2):158–173, Feb. 2001.

[16] R. Schlagenhaft, M. Ruhwandl, C. Sporrer, and H. Bauer. Dynamic load balancing of a multi-cluster simulator on a network of workstations. In *Proceedings of 9th Workshop on Parallel and Distributed Simulation (PADS '95)*, volume 25, pages 175–181. IEEE Computer Society, June 1995.

[17] A. Sloman and R. Poli. Sim agent: A toolkit for exploring agent designs. In M. T. Mike Wooldridge, Joerg Mueller, editor, *Intelligent Agents (ATAL-95)*, volume 2, pages 392–407. Springer-Verlag, 1996.

[18] K. Sycara. Multiagent systems. *AI Magazine*, 10(2):79–93, June 1998.

[19] G. Theodoropoulos and B. Logan. An approach to interest management and dynamic load balancing in distributed simulation. In *Proceedings of the 2001 European Simulation Interoperability Workshop*, pages 565–571, July 2001.

[20] L. Wang, S. Turner, and F. Wang. Resolving mutually exclusive interactions in agent based distributed simulations. In *Proceedings of the 1999 Winter Simulation Conference*, pages 1602–1609, 1999.

[21] E. White and M. Myjak. A conceptual model for simulation load balancing. In *Proceedings of the Spring Simulation Interoperability Workshops*, March 1998.

[2] `http://www.cs.bham.ac.uk/research/pdesmas`