Tuning Java to run interactive multiagent simulations over Jason[⋆]

Víctor Fernández-Bauset, Francisco Grimaldo, Miguel Lozano , and Juan M. Orduña

Computer Science Department, University of Valencia,
Dr. Moliner 50, (Burjassot) Valencia, Spain
{ferbau@alumni.uv.es,francisco.grimaldo, miguel.lozano, juan.orduna}@uv.es

Abstract. Java-based simulation environments are currently used by many multiagent systems (MAS), since they mainly provide portability as well as an interesting reduction of the development cost. However, this kind of MAS are rarely considered when developing interactive applications with time response constraints. This paper analyses the performance provided by Jason, a well-known Java-based MAS platform, as a suitable framework for developing interactive multiagent simulations. We show how to tune both the heap size and the garbage collection of the Java Virtual Machine in order to achieve a good performance while executing a simple locomotion benchmark based on crowd simulations. Furthermore, the paper includes an evaluation of Jason's performance over multi-core processors. The main conclusion derived from this work is that, by means of Java tuning, it is possible to run interactive MAS programmed using Jason.

1 Introduction and Related work

MAS platforms capable of handling a large amount of complex autonomous agents at interactive response times are required by interactive multiagent applications such as crowd simulations and massive online games. Usually, these kinds of simulations involve a high number of agents (e.g. pedestrians) interacting in a shared environment. Interactivity, in turn, requires the use of parallel techniques that allow to validate and to execute the actions requested within a limited period of time (commonly, 250 ms [6]).

Java-based simulation environments are currently being used by many MAS, since they mainly provide portability as well as an interesting reduction of the development cost. However, this kind of MAS are rarely considered when developing interactive applications with time response constraints, because of Java being normally less efficient than other languages such as C or C++. This situation requests performing a specific Java tuning to be able to tackle this type of

[⋆] This work has been jointly supported by the Spanish MEC and the European Commission FEDER funds, under grants Consolider-Ingenio 2010 CSD2006-00046 and TIN2009-14475-C04-04.

applications. In this paper, we show the Java tuning carried out for the purpose of evaluating the performace of Jason [1], a well-known Java-based MAS platform. The aim of this tuning is to adjust both the heap size and the garbage collection of the Java Virtual Machine in order to satisfy the temporal requirements of interactive multiagent simulations. Therefore, the results presented in this paper will also be of great value to those researches considering Java-based simulation environments suitable for developing interactive multiagent applications.

When developing this kind of interactive MAS three layers are normally considered: the computer architecture, the MAS platform and the graphical engine (if any). At the low level, different distributed computer architectures have been applied in order to allow massive interactive simulations to scale up with the number of agents by simply adding new hardware (e.g. networked-server, P2P, etc.). For instance, a new approach has been presented for PLAYSTATION3 which supports simulation of simple crowds of up to 15000 individuals at 60 frames per second [11]. Parallel simulation, based on classical Reynolds's boids [12], has been also integrated in a PC-Cluster with MPI communication [16] to finally produce small simulations (512 boids). At the top level, the graphical engine of the application must render the visualization at interactive frame rates. The computer graphics community generally represents the MAS as a particle system with local interactions [3, 15], though, few works include socially complex and autonomous behaviors [10]. However, they are not normally based on standard agent architectures.

In the middle level, the MAS platform is in charge of providing the required data flow to the graphical engine while efficiently using the computational resources. Thus, it constitutes a key middleware that highly influences the global performance and the scalability of the system. It mainly addresses two important issues: modeling the behavior of the agents as well as their parallel lifecycle execution. Java is a popular language providing built-in support for concurrency that is commonly used by MAS platforms. Although Java performance has been studied from different perspectives, probably the most usual is to tune server applications running on large multi-processor servers [13]. There are more specific works focused on the evaluation of Java-based multiagent platforms [2, 14, 8]. However, none of them deals with providing interactivity to the corresponding MAS. Some researchers have been also testing the performance and scalability of a few existing MAS platforms [7], showing a lack of both important issues in many of them. In a previous work [5], the authors analysed Jason's architecture and evaluated its performance under both centralised and distributed infrastructures. Regardless the infrastructure, the results showed that the execution options had to be reviewed in order to achieve a more equilibrated response time distribution, an aspect that we have covered in this work.

The rest of the paper is organized as follows. Section 2 briefly reviews Jason's centralised infrastructure and describes the locomotion benchmark used for the evaluation. Section 3 demonstrates how to tune Java in order to run interactive multiagent simulations over Jason. Finally, section 4 shows the performance obtained with different multi-core processors.

2 Test description

The goal of this work is to evaluate Jason as a suitable framework for running interactive multiagent simulations. Jason is a Java-based interpreter for an extended version of AgentSpeak, a BDI agent-oriented logic programming language [1]. Jason provides three infrastructures to execute a MAS: *Centralised*, SACI and JADE. Whereas the *Centralised* infrastructure places all the components of the MAS in the same host, it is also possible to distribute these components in several hosts using either SACI or JADE technologies. For the sake of simplicity, this paper focuses on the *Centralised* infrastructure but the results obtained are fully applicable for both distributed infrastructures.

In the Jason's *Centralised* infrastructure, the environment has its own execution thread and it is provided with a configurable pool of threads (PThE) devoted to executing the actions requested by the agents. In this way, the enviroment is able to deal with several agent requests concurrently. In turn, each agent owns by default a thread in charge of executing the agent reasoning cycle. In this manner, all the agents can run concurrently within the MAS. As such, this approach could limit the number of agents that can be executed, since the total number of threads would be limited by the Java Virtual Machine (JVM) heap size. However, Jason offers the possibility to optionally add another configurable pool of threads (PThA), so that the set of agents can share a smaller number of execution threads but reducing the level of concurrency. The number of threads in both PThE and PThA is initialised during the start-up of the MAS and it is not changed along its execution. By default, the PThE holds 4 threads whereas the PThA is disabled, so that each agent will have its own execution thread. In a previous work, we tuned both the PThE and the PThA in order to obtain the best performance [5].

The main issue to be tackled when running interactive multiagent simulations is that of being able of efficiently handling a massive and concurrent action processing. In this paper, we have used a locomotion testbed. Here, a set of wanderer agents request movement actions to a grid-like environment, which replies with the result of the execution. Wanderer agents are written in AgenSpeak and they cyclically execute the following steps: (i) take start time, (ii) request a random movement to the enviroment, and (iii) take finish time. On the other hand, the environment executes each movement action in a synchronized manner to ensure the world consistency. That is, the environment performs a simple collision test and informs whether the action can be carried out (i.e. Ok) or it cannot (i.e. Failure), when it would lead to a collision situation.

The performance evaluation carried out along the paper measures the environment response time and the percentage of CPU utilization consumed while running the locomotion benchmark. These measurements represent respectively latency and throughput, the two performance parameters commonly considered when evaluating networked-based distributed computer platforms [4]. We define the Response Time (*RT*) as the time elapsed between an agent asking for an action and receiving the reply from the environment. Our simulations stop when all the agents have performed 500 movements or cycles, but we discard the first

200 cicles when computing the average response time ($\overline{RT}$). Thus, we measure the system behavior at full load, since the first measurements are distorted due to the agent creation phase.

As stated above, we are interested in exploring the performace of Jason's *Centralised* infrastructure in depth. Thus, both the environment and the agents are run on the same host. The results for the *Centralised* infrastructure shown in [5] indicated that, when simulating 1000 wanderer agents, the 70% of the agents were able to act within 85 ± 264 ms. That is, even though the low value of $\overline{RT}$ (85 ms) indicated that many actions were processed very fast, there were a few agents that must wait more than 250 ms for their actions to be executed. This problem with the high standard deviation of the response time (σ_{RT}), found all over the measures in [5], is addressed in the following section.

3 Java tuning

The source of the high standard deviation of the response time of Jason-based MAS can be envisoned in figure 1. The figure shows that the average response time per agent cicle ($\overline{RT_c}$) peaks periodically. This points to a process that stops the system whenever it is executed: the Java Garbage Collection. Thus, we have carried out Java Performance Tuning in order to provide some general recommendations for running interactive multiagent simulations over Jason. It should be noticed, though, that the optimal tuning parameters will finally depend on the application and on the hardware underneath.

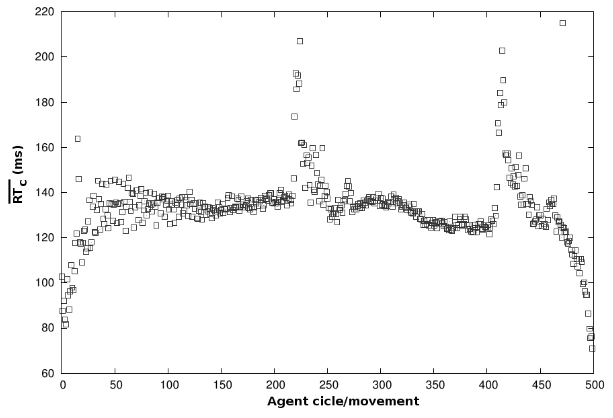

Fig. 1. Influence of the Java Garbage Collection on the response time

In this section, we show the results obtained when executing the testbed defined in section 2 over an AMD Dual-Core Opteron processor with 4 Gb of RAM, running a 64-bit version of Linux and the Sun's HotSpot^{TM}Java Virtual Machine (JVM) release 1.6.0_07. From version 1.5, this JVM has incorporated a technology to begin to tune itself, referred to as Ergonomics. Even though Ergonomics significantly improves the performance of many applications, optimal results often require manual tuning.

There are two main aspects that have to be tuned in order to enhance Java performance: the heap size and the garbage collector (GC) [9]. Regarding the former, by default, the initial heap size is 1/64th of the machine's physical memory and the maximum heap size is 1/4th of the machine's physical memory. In our case, this would mean using 64 Mb and 1 Gb respectively. However, Java performance can be enhaced by increasing the maximum heap size, as shown in figure 2. This figure shows the total amount of time consumed by the garbage collection when we use diferent GCs and increase the heap size while simulating 2500 agents. This time is computed by adding the times needed to complete every invocation to the GC. Besides, we have set minimum and maximum heap sizes equal for a faster startup. Note how, regardless of the GC being used, the total GC time strongly decreases when increasing the heap size up to 2 Gb. Further on, the gain is very low compared to the fact of being using almost the whole physical memory.

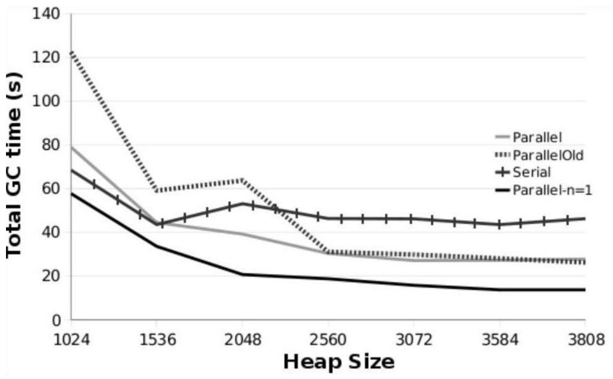

Fig. 2. Garbage colletion time needed for different heap sizes and GCs.

With respect to the garbage collectors, Sun's HotSpot^{TM}JVM allows the programmer to choose among three of them: *serial*, *throughput* and *concurrent* low pause collector. Whereas the *serial* GC is a sequential collector, the *throughput* GC uses multiple threads to collect garbage in parallel and it is suitable for applications with a large number of threads allocating objects, such as the one being tested in this paper. On the other hand, the *concurrent* GC does most of the collection concurrently with the execution of the application and it is appropriate for applications that benefit from shorter GC pauses. Additionally, Java GCs organize the object memory into two generations: *young* (recently created objtects) and *tenured* (older objects). Java allows to set the ratio between the *young* and *tenured* generation by means of the JVM command-line option *NewRatio*. For more details on Java garbaje collection, see [9].

Bearing all this informacion in mind, we have executed our benchmark using every GC available. Figure 2 shows the most relevant results that we have obtained. The line named *Serial* corresponds to the total amount of time consumed by the garbage collection when simulating 2500 agents using the *serial* GC. The *Parallel* line relates to the use of the *throughput* GC only for the collection of the *young* generation. In turn, the *ParallelOld* line refers to the use of the *throughput* GC for the collection of both the *young* and the *tenured* generation. For space reasons, we skip the results obtained with the *concurrent* GC since they are up to ten times higher than those obtained with the rest of the GCs, both for the total GC time and for the average response time. As we can observe, the *serial* GC behaves worse than any configuration of the *throughput* GC. Moreover, parallelizing the collection of the *tenured* generation does not fasten but actually slows garbage collection when the heap size is less than 2.5 Gb. This means that there is not a problem with the collection of old objects but with the young ones. The reason behind this fact relies on how Jason represents agent's beliefs and actions. Both are implemented as objects that are discarded and created again whenever there is a change in a belief or a new action is requested to the environment. As each wanderer agent continuously asks the environment for movement actions and changes its position, we can imagine the huge amount of objects that "die young". Thus, enlarging the *young* generation will benefit garbage collection.

The default *NewRatio* for the Server JVM is 2. That is, the *tenured* generation occupies 2/3 of the heap while the *young* generation occupies 1/3. A larger *young* generation could accommodate many more short-lived objects, decreasing the need for slow major collections. Meanwhile, the *tenured* generation would still be large enough to hold many long-lived objects. According to this, the line labeled as *Parallel-n=1* in figure 2 shows that we can obtain the lowest garbage collection times by using the *throughput* GC for the collection of the *young* generation along with the minimum ratio possible between the generations (i.e. $NewRatio = 1$). Hence, half of the heap for the *young* generation and the other half for the *tenured* generation.

Finally, we have evaluated the effect of the number of threads devoted to collect garbage when using the parallel *throughput* GC. By default, this GC uses as many garbage collector threads as the number of processors available. Though, the number of threads can be tuned manually through the *ParallelGCThreads* command-line option. For this test, we have used a 16-core computer and we have varied the number of collector threads from 2 up to 16. Besides, we have tuned Java so it runs efficiently with 2 Gb of heap size and the *NewRatio* set to 1. Figure 3 shows the values obtained for the average response time ($\overline{RT}$) plus its standard deviation (σ_{RT}) when increasing the number of agents simulated. Evidently, the worst values are obtained when only 2 threads are used for garbage collection. However, in our test it is not necessary to use as many threads as the number of cores, since we get the same results for 8 and 16 GC threads.

Summing up, we can state the following general recommendations for running interactive multiagent simulations over Jason:

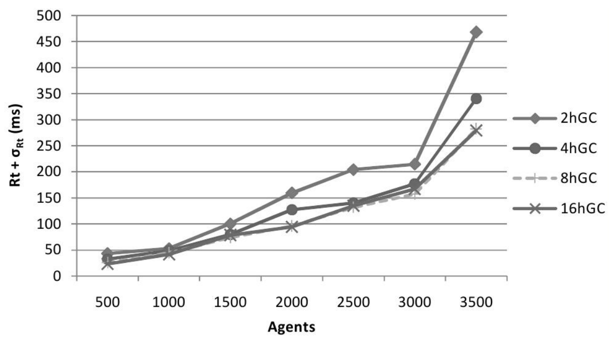

Fig. 3. Performance when varying the number of threads used by the *throughput* GC

– Enlarge the heap size as much as possible without achieving the amount of physical memory available. In addition, set minimum and maximum heap sizes equal for a faster startup.
– Parallelize garbage collection by using the *throughput* GC whenever your hardware has at least 2 CPUs in order to reduce GC pause times. Besides, check whether you need the default number of collector threads (equal to the number of processors) or you can save any, thus reducing the workload of the whole machine.
– Increase the size of the *young* generation up to the size of the *tenured* generation (*NewRatio*=1) to decrease the need for slow major collections.

4 Performance Evaluation

In this section we analyse the results obtained when running the benchmark described in section 2 on the following distributed shared memory (DSM) multi-core computers: 2-Core (AMD Dual-Core Opteron, 1.6 GHz, 4 GB RAM), 4-Core (AMD Quad-Core Opteron, 1.0 GHz, 8 GB RAM), 8-Core (Intel 8-Core Xeon, 2.6 GHz, 16 GB RAM) and 16-Core (AMD Dual-Core 8218, 1.0 GHz, 32 GB RAM). All of them run the same 64-bit version of Linux and the Sun's HotSpot^{TM}JVM release 1.6.0_07.

Table 1 shows the performance obtained when simulating from 1500 to 9500 wanderer agents on the computers described above. The results for 1-core were obtained through the *taskset* Linux command. When running the benchmark, we have followed the Java tuning recommendations stated in section 3. Therefore, we have used the *throughput* GC for the collection of the *young* generation with a number of collector threads equal to the number of cores. Besides, we have tuned Java so it runs with 4 Gb of heap size and we have set *NewRatio* to 1. The left column in Table 1 shows the percentage of CPU utilization measured during the execution of the simulation. The central column ($\overline{RT}$) shows the average Response Time in milliseconds for the actions requested by the agents when the system is at full load, as explained in section 2. Finally, the right column shows the standard deviation of this Response Time (σ_{RT}).

Cores-Agents	CPU(%)	$\overline{RT}$(ms)	σ_{RT}(ms)	Cores-Agents	CPU(%)	$\overline{RT}$(ms)	σ_{RT}(ms)
1-1500	89,53	44,59	101,64	8-1500	59,88	31,01	8,58
1-3500	90,01	40,39	189,57	8-3500	67,75	73,82	22,65
1-5500	89,98	71,97	178,42	8-5500	72,09	114,10	40,62
1-7500	87,87	85,93	193,03	8-7500	74,56	146,27	58,26
1-9500	65,97	98,33	2196,68	8-9500	74,92	185,81	278,00
2-1500	89,17	3,92	28,84	16-1500	39,77	57,38	9,60
2-3500	91,13	5,55	27,59	16-3500	46,45	145,86	38,10
2-5500	92,00	9,01	35,38	16-5500	48,27	242,87	62,23
2-7500	91,10	10,39	79,09	16-7500	57,58	282,57	85,73
2-9500	59,72	47,79	1152,10	16-9500	57,51	253,53	534,66
4-1500	76,25	51,97	20,81				
4-3500	81,11	132,88	50,71				
4-5500	81,48	201,90	76,89				
4-7500	83,35	290,71	118,30				
4-9500	84,24	386,35	488,37				

Table 1. Performance obtained for Jason framework over different computers

The results shown in Table 1 demonstrate that we can run interactive multiagent simulations over Jason, since the values of the $\overline{RT}$ plus the σ_{RT} are generally under the reference value of 250 ms. As it was also expected, the CPU utilization decreases as the number of cores increases. For instance, if we compare the results obtained for 3500 agents on each computer, it can be seen that the more cores in the computer, the lower the percentage of CPU utilization (the single CPU is shown only as a reference). However, the response time does not behave the same way. Instead, whereas the $\overline{RT}$ values for the 2-Core computers are around a few milliseconds, the $\overline{RT}$ for the computers with 4 up to 16 cores reaches tens of milliseconds. The worsening of the response time occurs in all the computer being tested, although it has a minor impact in the 8-Core computer because it has the highest processor speed. This fact indicates that, beyond two cores, the default configuration used by Jason does not properly scale up with the number of processor cores. Thus, a deeper study must be carried out in order to allow it to take advantage of the multi-core processors.

Although a fine tuning of the Jason framework for multi-core processors is beyond the scope of this paper, we have analysed the issue shown in Table 1 in order to clarify the path for future work. We think that the reason behind this problem is thread context switching. Even though the Java Virtual Machine schedules its threads to run them as fast as possible, there is no guarantee of which core a given thread will be assigned to for execution. The operating system kernel can assing one single thread to different cores during its execution time, thus provoking thread migrations. The probability of migration increases with the number of cores in the processor, in such a way that the overhead due to thread migrations could exceed the benefits of having more cores for executing the threads in parallel. To verify this hypothesis, we have measured the number of migrations (i.e. changes in the core assigned for execution) suffered by the threads along the simulation. To detect migrations, we have used a system call retrieving the state of the Java threads periodically and we have analysed the core where they were located.

Figure 4 shows the total number of thread migrations counted while executing the same simulations that produced the results of Table 1. We can observe how the number of migrations is proportional to the number of cores in the computer. Since a thread migration is a time consuming task, the high number of migrations produced by computers with more than 2 cores can explain the behavior shown in Table 1. Nevertheless, it should be noticed that these results do not guarantee the absence of other still hidden aspects that could prevent the system from properly scaling with the number of processor cores. In order to fully exploit the degree of parallelism offered by multi-core processors, tuning the processor affinity of Jason must be done.

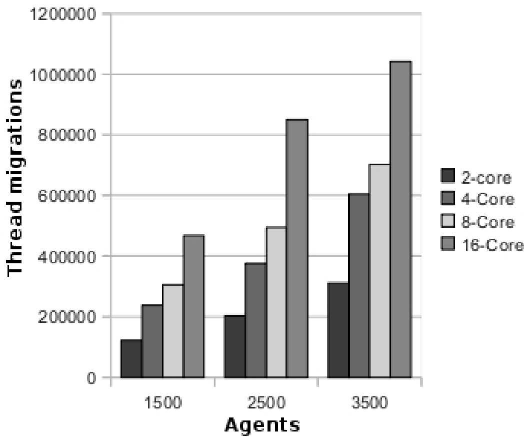

Fig. 4. Number of thread migrations

5 Conclusions and Future work

In this paper, we have evaluated Jason as a suitable Java-based MAS platform for developing interactive multiagent simulations. We have shown how to tune the Java heap size as well as the gargabe collector in order to enhance the performance of the simulations. Even though the optimal tuning parameters will finally depend on the application and on the hardware underneath, we have state some general recommendations for minimizing the impact of garbage collection. Therefore, the results presented in this paper will also be of great value to those researches considering other Java-based simulation environments for developing interactive multiagent applications. The paper also includes a first evaluation of Jason's performance over multi-core processors. As future work, we plan to carry out a deep study of the Jason framework in order to properly scale it up with the number of processor cores. Then, tuning the Java processor affinity will be required to exploit the degree of parallelism offered by multi-core processors.

References

1. R. H. Bordini, J. F. Hübner, and M. Wooldrige. *Programming Multi-Agent Systems in AgentSpeak using Jason.* Wiley, 2007.
2. E. Cortese, F. Quarta, and G. Vitaglione. Scalability and performance of JADE message transport system. In *AAMAS Workshop on AgentCities*, 2002.
3. S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan. Geopostors: a real-time geometry/impostor crowd rendering system. *ACM Trans. Graph.*, 24(3):933–933, 2005.
4. J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach.* Morgan Kaufmann, 2002.
5. V. Fernández, F. Grimaldo, M. Lozano, and J. M. Orduña. Evaluating Jason for distributed crowd simulations. In *Proc. of the 2nd. International Conference on Agents and Artificial Intelligence*, volume 2, pages 206–211. INSTICC Press, 2010.
6. M. Lozano, P. Morillo, J. M. Ordua, and V. Cavero. On the design of an efficient architercture for supporting large crowds of autonomous agents. In *Proceedings of IEEE 21th. International Conference on Advanced Information Networking and Applications (AINA'07)*, pages 716–723, 2007.
7. L. Mulet, J. M. Such, and J. M. Alberola. Performance evaluation of open-source multiagent platforms. In *Proc. of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1107–1109. ACM, 2006.
8. C. Nikolai and G. Madey. Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation*, 12(2):2, 2009.
9. Oracle Sun Depeloper Network. Java Tuning White Paper, 2010. Available at http://java.sun.com /performance/reference/whitepapers/tuning.html.
10. N. Pelechano, J. M. Allbeck, and N. I. Badler. Virtual crowds: Methods, simulation, and control. *Synthesis Lectures on Computer Graphics and Animation*, 3(1):1–176, 2008.
11. C. Reynolds. Big fast crowds on ps3. In *Proc. of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121. ACM, 2006.
12. C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proc. of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34. ACM, 1987.
13. J. Shirazi. *Java Performance Tuning.* O'Reilly, 2003.
14. R. Tobias and C. Hoffman. Evaluation of free java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation*, 7(1), 2004.
15. A. Treuille, S. Cooper, and Z. Popovic. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1160–1168. ACM, 2006.
16. B. Zhou and S. Zhou. Parallel simulation of group behaviors. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 364–370. Winter Simulation Conference, 2004.