Exploiting D-Mason on Parallel Platforms: A Novel Communication Strategy

Gennaro Cordasco[1], Francesco Milone[2],
Carmine Spagnuolo[2], and Luca Vicidomini[2]

[1] Dipartimento di Psicologia, Seconda Università degli Studi di Napoli, Italy
gennaro.cordasco@unina2.it
[2] Dipartimento di Informatica, Università degli Studi di Salerno, Italy
milone.francesco1988@gmail.com, {cspagnuolo,lvicidomini}@unisa.it

Abstract. Agent-based simulation models are a powerful experimental tool for research and management in many scientific and technological fields.

D-Mason is a parallel version of Mason, a library for writing and running Agent-based simulations.

In this paper, we present a novel development of D-Mason, a decentralized communication strategy which realizes a *Publish/Subscribe* paradigm through a layer based on the MPI standard. We show that our communication mechanism is much more scalable and efficient than the previous centralized one.

Keywords: Publish/Subscribe, MPI, Agent-based simulation models, Mason, D-Mason, Parallel Computing, Distributed Systems, High Performance Computing.

1 Introduction

Agent-Based Model (ABM) denotes a class of models which, simulating the behavior of multiple agents (i.e., independent actions, interactions and adaptation), aims to emulate and/or predict complex phenomena.

Successes in Computational Sciences over the past ten years have caused increased demand for supercomputing resources, in order to improve the performance of ABMs in terms of both number of agents and complexity of interactions.

Parallel computing has becoming the dominant paradigm for computational scientist (indeed, serial-processing speed is reaching a physical limit [15]). Unfortunately, exploiting parallel systems is not an easy task: performance has to be realized through concurrency, with applications designed to scale as the number of resources increases.

Computer science community has responded to the need for tools and platforms that can help the development and testing of new models in each specific field by providing tools, libraries and frameworks that speed up and make easier the task of developing and running parallel ABMs for complex phenomena.

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 407–417, 2014.
© Springer International Publishing Switzerland 2014

D-Mason [6,16] is a parallel version of the Mason [3,10,11] library for running ABMs on distributed systems. D-Mason adopts a framework-level parallelization mechanism approach, which allows to harness the computational power of a parallel environment and, at the same time, hides the details of the architecture so that users, even with limited knowledge of parallel computer programming, can easily develop and run simulation models.

In [7] a preliminary discussion about the use of MPI primitives for the development of a Publish/Subscribe (PS) service has been showed. This paper makes a step forward in that direction; we present a novel communication strategy, based on the PS paradigm, which uses the MPI Standard [17] as an example of distributed communication on D-Mason.

After a brief review and a critical analysis of the state of the art of D-Mason (Section 2), we report, in Section 3, the details of the novel MPI Publish/Subscribe layer which: (i) improves the preliminary version (cf. [7]); (ii) provides also a hybrid approach exploiting the advantages of both the centralized and decentralized communication strategies. Finally, in Section 4 we report an extensive set of experiments showing that the novel MPI-based Publish/Subscribe mechanism is extremely advantageous when the number of computing machines is large. In this case, in fact, a single communication server is unable to handle all the communication that the system requires and thus it represents a bottleneck for the whole system.

2 Mason and D-Mason

Mason toolkit is a discrete-event simulation core and visualization library written in Java, designed to be used for a wide range of ABMs. Mason is based on a standard Model-View-Controller (MVC) paradigm and three layers compose it: the *simulation* layer, the *visualization* layer and the *utility* layer.

D-Mason adds a new layer named D-Simulation, which extends the Mason simulation layer. The new layer adds some features to the simulation layer that allow the distribution of the simulation workload on multiple, even heterogeneous, machines. The intent of D-Mason is to provide an effective and efficient way of parallelizing Mason simulations: effective because with D-Mason you can do more than what you can do with Mason, efficient because the porting of an application from Mason to D-Mason happens with some incremental modifications to the Mason application without the need of re-designing it.

D-Mason is based on a Master-Worker paradigm: some workers, henceforth logical processors (LPs), perform the simulation while a master application is in charge of: discovering the LPs, bootstrapping the system, managing and interacting with the simulation. D-Mason adopts a space partitioning approach where the space to be simulated (D-Mason's field) is partitioned into regions. Each region, together with the agents contained in it, is assigned to a LP. Since usually the area of interest (AOI) of an agent is small compared with the size of a region, the communication between workers, required to synchronize the simulation step by step, is limited to local messages (messages between LPs, managing neighboring spaces, etc.).

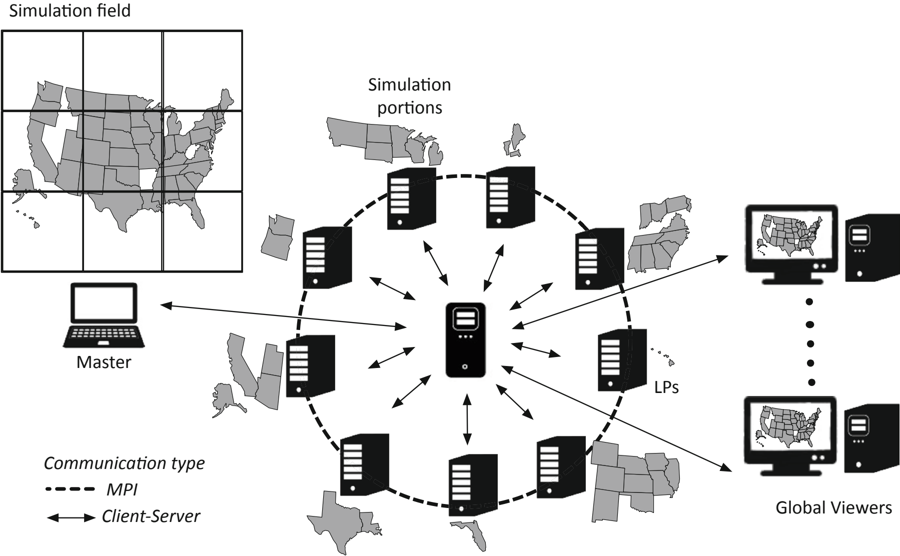

Fig. 1. D-Mason scheme

In a discrete-event simulation, events need to be processed in a non-decreasing timestamp order, because an event with a smaller timestamp can potentially modify the state of the system and thereby affect events that happen later. We call this phenomenon *causality constraint*. On a sequential simulation, the causality constraint is easily satisfied by using a queue of events ordered by timestamp. On parallel simulation, the problem is much tougher and two main approaches have been introduced to deal with it: *Optimistic approach,* which allows events to be processed out of order. Once a causality error is detected, the offending LP has to rollback and recover from such an error. This kind of approach requires state saving and recovery mechanisms [9]; *Conservative approach* which guarantee that events are always processed in the right order. D-Mason adopts a conservative approach: each simulation step is divided in two phases: *communication/synchronization* and *simulation*. Each simulation step is associated with a fixed state of the simulation. Regions are simulated step by step: the step i of a region r is computed according to the states $i-1$ of r's neighboring regions, so the step i of a region cannot be executed until the states $i-1$ of its neighbors have been computed and delivered. This approach does not need any rollback strategies but each simulation step represents a barrier; the system advances with the same speed provided by the slower LP in the system. For this reason, it is necessary to balance the load among workers.

Figure 1 depicts the architecture of D-Mason.

Current Centralized Communication Strategy in D-Mason with ActiveMQ

D-Mason uses a well-known communication mechanism, based on the Publish/Subscribe (PS) design pattern, to propagate agents' state information: a multicast channel is assigned to each region; LPs then simply subscribe to the channels associated with the regions, which overlap with their AOI to receive relevant message updates.

The first versions of D-Mason used Java Message Service (JMS) [8] for communication between workers. A dedicated machine that runs an Apache ActiveMQ Server [1] and acts as a JMS provider (i.e., it allows to generate and manage multicast channels and route messages accordingly) was used. D-Mason however, is designed to be used with any Message Oriented Middleware that implements the PS pattern.

The choice for a centralized dedicated communication service was due to the fact that D-Mason was initially conceived to harness the amount of unused computing power available in common installations like educational laboratories. In this setting, the choice for a dedicated communication server was preferred for several reasons. It does not require the installation of a specific communication middleware on each logical processor. All communication is handled by a single machine, consequently all the computational power provided by LPs is dedicated to simulation phase. The number of machines (LPs) available in common laboratory is limited, therefore the centralized communication does not represent a bottleneck for the system, as confirmed by the experiments in [6].

The More You Get, the More You Want

Considering the good results obtained by D-Mason we wondered if the approach used by D-Mason (a framework-level parallelization mechanism) could also be exploited for dedicated installation, such as massively parallel machines and clusters of workstations. If so, what changes are needed in order to adapt D-Mason for dedicated installation? These platforms usually offer a large number of homogeneous machines that, on one hand, simplify the issue of balancing the load among LPs [4], but, on the other hand, the considerable computational power provided by the system weakens the efficiency of the communication server. Indeed, centralized solutions cannot scale with the growth of the computational power (which affects the amount of communication) and especially in the number of LPs (number of communication).

The main goal of our paper is to check whether the communication strategy in D-Mason architecture can be improved using a distributed MPI layer.

3 Decentralized Communication Strategy in D-Mason with MPI

MPI is a library specification for message-passing, designed for high performance on both massively parallel machines and on workstation clusters. MPI

has emerged as one of the primary programming paradigms for writing efficient parallel applications; it provides point-to-point and collective communications and guarantees portability with all platforms compliant with the MPI Standard. MPI provides several collective operations, which are very important because they sustain very high parallel speed-ups for parallel applications [17]. Our implementation is based on *mpiJava*, a Java binding of MPI-1.1 Standard [2, 5].

A Distributed MPI Publish/Subscribe Layer

The communication model in D-Mason is potentially n-to-n, which means that each LP of the network may need to communicate with all others. D-Mason is based on the Publish/Subscribe paradigm to meet the requirements of flexibility and scalability of the system. In more details, the Communication Layer of D-Mason exploits the flexibility of the Publish/Subscribe paradigm to virtualize groups of communication between the agents. In the distributed simulation, these groups communicate at the end of each simulation step.

MPI does not provide Publish/Subscribe functionalities so we had to develop a different layer, according to the communication interface of D-Mason, which exposes some routines to publish and receive messages on specific topics. This layer is based on MPI collective communications (i.e., MPI_Bcast and MPI_Gather) which allows making a series of point-to-point communications in one single call. MPI processes can be grouped and managed by an object called *Communicator* [13].

The JMS Strategy and the MPI one handle the synchronization in a different way. In the JMS strategy the synchronization is implemented at the framework level using a data structure that indexes, for each step, the updates and acts as barrier, so that each cell remains locked until it receives all updates. In the MPI strategy, we take advantage of the intrinsic synchronization of MPI, because the collective communication primitives are blocking.

In [7] the details of three different implementations have been presented: *MPI_Bcast*, *MPI_Gather* and *MPI_Parallel*. The first two strategies are based on the MPI group communication primitives of the same name and are almost equivalent in terms of performances in real scenarios, while *MPI_Parallel* allows us to increase the degree of parallelism during the synchronization phase, resulting in increased performances.

The parallel strategy is based on the following considerations. Each synchronization phase requires a certain set C of communication where each communication is identified by a pair $\langle$sender, receiver$\rangle$. Using MPI a set of communication can be executed in parallel provided that each process appears at most once (either as sender or as receiver). Hence, we need to partition C in such a way that each set obtained can be executed in parallel and the number of sets is as small as possible. This problem is a well-known NP-Hard problem: *Edge coloring* [12]. An edge coloring of a graph is a minimum assignment of colors to the edges of the graph so that no adjacent edges have the same color. In [7] a simple randomized heuristic was presented to find a good partition in a reasonable time.

Implementation

A preliminary implementation of MPI Publish/Subscribe pattern was provided in a previous work [7]. In this work, we updated the implementation according to the latest Java binding available in OpenMPI V. `1.7.5`.

This required, in fact, a major rewrite of our implementation. Briefly, the previous Java binding relied upon the `MPI.Object` class in order to automatically perform (de-)serializing of arbitrary Java objects. This feature was removed in OpenMPI V. `1.7.5` so we had to manually perform (de-)serializing in order to send arrays of `MPI.Byte` objects.

The package `dmason.util.connection` provides the interface `Connection`, which defines the Publish/Subscribe functionalities. In this new version, D-Mason's communication layer offers three implementations: one is based on *Apache ActiveMQ*, one on *MPI* and one, named *hybrid*, uses both ActiveMQ and MPI. Specifically, the hybrid implementation uses MPI for 1-to-n communications between the system management and the LPs and for n-to-1 communications between the LPs and the visualization component, while it uses MPI for the simulation updates between LPs (synchronization). Both the implementations that exploit MPI have been implemented using two out of the tree strategies described above: MPI_Bcast and MPI_Parallel. The latter strategy is highly recommended when running simulations with a large number of LPs.

In Section 4 we show a performance analysis of the new D-Mason's decentralized communication layer.

4 Results

We analyzed the performance of the novel D-Mason communication layer against the centralized ActiveMQ approach performing a number of tests on large simulations. Experiments have been carried on several configurations obtained varying several parameters: number of agents, fields dimension, AOI radius and number of regions. Such parameters determine a ratio between the communication and computation requirements. We expected that the benefits of the new strategy are proportional to the ratio communication / computation. Indeed using the centralized approach, the synchronization is handled by the ActiveMQ server, whereas using the decentralized approach the synchronization represents a computational cost for each LP. Hence, only when the ratio communication / computation is sufficiently large, that cost is paid off in terms of efficiency of communication.

We also evaluated the scalability and the effectiveness of latest implementation of D-Mason's communication layer in exploiting homogeneous hardware.

Setting and Goals of the Experiments

We have used a cluster of eight nodes, each equipped as follows:

- CPUs: 2 x Intel(R) Xeon(R) CPU E5-2680 @ 2.70GHz (#core 16, #threads 32)

- RAM: 256 GB
- Network: adapters Intel Corporation I350 Gigabit

Considering the high computational power of each node we were able to run several (up to 90) LPs on each node. Simulations have been conducted on a scenario consisting of seven machines for computation and one for managing the simulations and running the ActiveMQ server when needed.

We have tested the simulation *Flockers* available in MASON, an implementation of the well–known "*Boids*" model by Craig Reynolds [14], stated in 1986. We chose this simulation due to the embarrassingly parallelizable nature of the problem. Concisely, the *Flockers* model simulates the flocking behavior of birds and its relevant aspect is that the interactions are purely local between each agent and its neighbors; for such reason the simulation fits very well to the execution in a distributed environment.

We performed three categories of experiments:

1. *Communication scalability*: this test aims to evaluate the scalability of the communication layer in terms of the number of LPs. As the number of LPs increases, the communication requirements become crucial in the efficiency. On the other hand, on very large simulations the ability to run a large number of LPs is essential in order to partition the overall computation without exceeding the physical limits of each LP in the system;
2. *Computation scalability*: this test aims to evaluate the scalability of the communication layer in terms of the number of simulated agents. In this case an increase of the number of agents corresponds to an increase of the computational power required, and consequently to a reduction of the ratio communication / computation.
3. *Robustness*: this test aims to assess the effectiveness of the proposed solution on different scenarios.

Communication Scalability Test

For this experiment, we fixed both the field size ($10,000 \times 10,000$), the number of agents (1 million) and the AOI (10). We employ 16 test settings, each characterized by: the field partitioning configuration (number of rows and columns), which determines also the number of Logical Processes (Number of LPs = $[R]ows \times [C]olums$) and the communication scheme (MPI or ActiveMQ). A couple (P, S) identifies each test setting where

- $P \in \{2 \times 2, 3 \times 3, 4 \times 4, 5 \times 5, 10 \times 10, 15 \times 15, 20 \times 20, 25 \times 25\}$ is the field partitioning configuration.
- $S \in \{ActiveMQ, MPI\}$ is the communication scheme.

We compared the two communication schemes by running the simulation *Flockers* for $3,000$ simulation steps. Each simulation has been executed several times in order to check for any fluctuations in the results but we observed no significant changes.

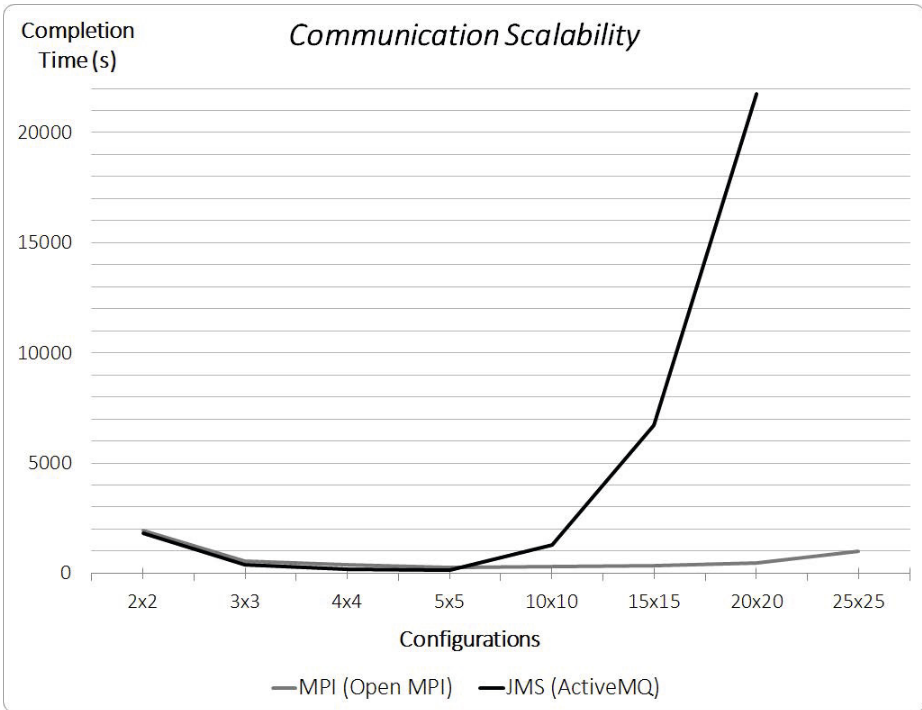

Fig. 2. Communication scalability

Figure 2 presents the results. The $X-$axis indicates the value of P (left to right the number of LPs is increasing), while the $Y-$axis indicates the overall execution time in seconds. Notice that there is a point missing because the test setting $(25 \times 25, ActiveMQ)$ crashes after few steps (the ActiveMQ server is not able to manage the communication generated by 625 LPs.)

When the number of LPs is small, the advantage of the decentralized communication does not appear because the message broker is much efficient comparing to the coarse grain synchronization requirement of the decentralized one. By increasing the number of LPs, the efficiency of the centralized message broker gets down dramatically and the simulation performance does exhibit the benefits of using the decentralized communication. This trend is due to the fact that by increasing the LPs number there are much more messages in the system and the effort needed to have a synchronizing mechanism in the decentralized communication approach is hidden by the time taken by the message broker to deliver all the messages.

Computation Scalability Test

For this experiment, we fixed the density of the field (i.e., field area divided by number of agents) and the AOI (10). We employ 72 test settings, each characterized

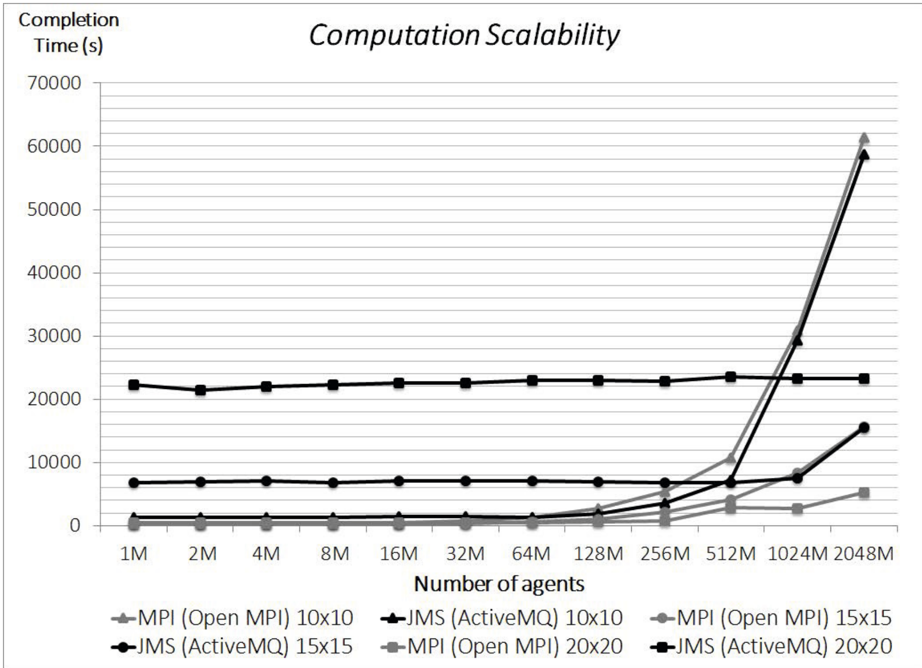

Fig. 3. Computation scalability

by: the field partitioning configuration, the communication scheme and the number of agents. Each test setting is identified by a triple (P, S, A) where

- $P \in \{10 \times 10, 15 \times 15, 20 \times 20\}$ is the field partitioning configuration.
- $S \in \{ActiveMQ, MPI\}$ is the communication scheme.
- $A \in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048\} \times 1,000,000$ (M) is the number of agents.

We compared six configurations, each one characterized by a field partitioning configuration and a communication scheme, by running the simulation *Flockers* for $3,000$ simulation steps.

Figure 3 presents the results. The $X-$axis indicates the number of agents A, while the $Y-$axis indicates the overall execution time in seconds. The test starts with a field size of $10,000 \times 10,000$ and one million of agents, these values are scaled up proportionally in such a way to keep a fixed density along the overall test.

The figure shows that for each field configuration the MPI approach performs better than ActiveMQ up to a certain number of agents (i.e., $64M$ for 10×10 configuration) that is when the computational requirement are significantly higher than the communication one. However, the figure shows also that if this is the case then the system deserves a finer field partitioning. Indeed, by increasing the number of LPs (i.e., moving from 10×10 to 15×15) we are able

to obtain better performances. Moreover increasing the number of LPs requires more communication, which increases the ratio communication / computation and consequently shifts the "*cross-point*" ($1024M$ for 15×15 configuration). We notice that in the last field configuration (20×20), the cross point has not been reached because the ActiveMQ server is not able to manage the communication generated by more than $2048M$ agents.

Robustness

We also study the robustness of the schedule by varying the parameters that have been fixed in previous experiments and checking that they do not affect the results in terms of efficiency.

For this experiment, we fixed both the field partitioning configuration (20×20) and the number of agents (1 million). We performed two different experiment by changing:

- Area of interest (AOI), this test observes the ability of the two different communication layers to manage different sizes of messages. We tested several values of the AOI, ranging from 5 to 80;
- Average field density (FD), the $FD = \frac{A}{W \times H}$, where A is the number of agents, W is the field's width and H the field's height. FD is the average density of agents within the field, obviously is possible to change this value setting different values of the field size or the number of agents (or both); varying the value of FD affect both the messages size and the computation requirements. We tested different FDs, changing the field size in a range from $5,000 \times 5,000$ ($FD = 1/25$) to $20,000 \times 20,000$ ($FD = 1/400$).

Both the experiments show the same trend in which the MPI solution clearly beats the centralized one. The improvement is always more than 95%. Because the new results are so close to the ones we have presented, there would be no value in exhibiting new plots.

5 Conclusions and Future Works

The performance results described in the previous Section show that the novel communication strategy allows taking advantage from using homogeneous hardware when the simulation requires a sensible amount of communication. As a future work, it would be interesting to devise a specific test in order to characterizing the communication / computation trade-off, that is to determine the minimum ratio communication / computation, beyond which the MPI approach is more efficient than the centralized one.

The *MPI* communication layer uses the Java binding of *MPI*, available in *OpenMPI*; during our work, we discovered some limits of this kind of solution as described in [7]. Moreover, the current stable binding is not tread safe.

The bindings is in continuous development following a JNI approach; mpiJava [2] was taken as a starting point for *OpenMPI* Java binding, but it was later totally rewritten.

The novel MPI approach provides also several by-products: First of all the synchronization among LPs is easier because MPI calls are blocking. Moreover, MPI provides also several features, such as *dynamic process creation and management*, which simplifies the management of the system, especially using heterogeneous hardware.

Finally, we are still working to enhance the efficiency of the MPI communication layer on either the communication strategy or the *mpiJava* implementation.

References

1. Apache ActiveMQ Server, `http://activemq.apache.org/`
2. Baker, M., Carpenter, B., Fox, G., Ko, S.H., Lim, S.: mpiJava: An object-oriented Java interface to MPI. In: Rolim, J., et al. (eds.) Parallel and Distributed Processing. LNCS, vol. 1586, pp. 748–762. Springer, Heidelberg (1999)
3. Balan, G.C., Cioffi-Revilla, C., Luke, S., Panait, L., Paus, S.: MASON: A Java Multi-Agent Simulation Library. In: Proceedings of the Agent 2003 Conference (2003)
4. Carillo, M., Cordasco, G., Chiara, R.D., Raia, F., Scarano, V., Serrapica, F.: Enhancing the Performances of D-MASON - A Motivating Example. In: SIMULTECH, pp. 137–143 (2012)
5. Carpenter, B., Fox, G.C., Ko, S.-H., Lim, S.: mpijava 1.2: API Specification (1999)
6. Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., Spagnuolo, C.: Bringing together efficiency and effectiveness in distributed simulations: The experience with D-MASON. SIMULATION: Transactions of The Society for Modeling and Simulation International 89(10), 1236–1253 (2013)
7. Cordasco, G., Mancuso, A., Milone, F., Spagnuolo, C.: Communication Strategies in Distributed Agent-Based Simulations: The Experience with D-Mason. In: an Mey, D., et al. (eds.) Euro-Par 2013. LNCS, vol. 8374, pp. 533–543. Springer, Heidelberg (2014)
8. Java Message Service Concepts, `http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html`
9. Liu, J.: Parallel Discrete-Event Simulation. Wiley Encyclopedia of Operations Research and Management Science (2009)
10. Luke, S., Cioffi-revilla, C., Panait, L., Sullivan, K.: MASON: A New Multi-Agent Simulation Toolkit. In: Proceedings of the 2004 SwarmFest Workshop (2004)
11. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: MASON: A Multiagent Simulation Environment. Simulation 81(7), 517–527 (2005)
12. Misra, J., Gries, D.: A Constructive Proof of Vizing's Theorem. Inf. Process. Lett. 41(3), 131–133 (1992)
13. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, TN, USA (July 1997)
14. Reynolds, C.W.: Flocks, Herds and Schools: A Distributed Behavioral Model. SIGGRAPH Comput. Graph. 21(4), 25–34 (1987)
15. Sutter, H.: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dobb's Journal 30(3) (2005)
16. D-MASON Official Website, `http://www.dmason.org` (accessed May 2014)
17. MPI Standard Official Website, `http://www.mcs.anl.gov/research/projects/mpi/index.htm` (accessed: April 25, 2013)