Scalable Computing: Practice and Experience
Volume 9, Number 2, pp. 127–134. http://www.scpe.org
ISSN 1895-1767
© 2008 SCPE

MOBILE AGENT SYSTEMS INTEGRATION INTO PARALLEL ENVIRONMENTS*

DAVID E. SINGH†, ALEJANDRO MIGUEL, FÉLIX GARCÍA, AND JESÚS CARRETERO

Abstract. In this work MASIPE, a tool for monitoring parallel applications, is presented. MASIPE is a distributed tool that gives support to user-defined mobile agents, including functionalities for creating and transferring these agents through different compute nodes. In each node, the mobile agent can access the node information as well as the memory space of the parallel program that is being monitored. In addition, MASIPE includes functionalities for managing and graphically displaying the agent data. In this work, its internal structure is detailed and an example of a monitored scientific application is shown. We also perform a study of the MASIPE requirements (in terms of CPU and memory) and we evaluate its overhead during the program execution. Experimental results show that MASIPE can be efficiently used with minimum impact on the program performance.

Key words: program monitoring, mobile agents, distributed system.

1. Introduction. Nowadays, the increasing computational power of commodity computers and interconnection networks allows efficiently executing parallel applications in low-cost platforms. Commodity computers can be organized conforming clusters which are usually specifically designed for CPU-intensive computing. For this configurations, there are several tools [1, 2, 3, 4] that provide solutions for system management. However, commodity computers can also be organized in labs or computer rooms, usually as single-user, stand-alone computers. In this scenario, CPU-intensive computing is a secondary task. Under these circumstances, the use of cluster management tools is complicated, given that they can interfere with the front-end user applications. For example, it would be necessary to explicit start the monitor tools in each compute node or performing a system reboot for starting a different operative system. Consequently, in this kind of environments it is more interesting to use non-intrusive tools for monitoring parallel programs. These tools should be automatically executed, using the same execution environment as the one employed with parallel application.

The work presented in this paper addresses this problem by means of a new monitoring tool called *Mobile Agent Systems Integration into Parallel Environments* (MASIPE). This tool was presented in [5] and this paper is an extension of the work presented there. MASIPE is designed for monitoring parallel applications by means of mobile agents. It provides a platform for executing agent-based programs in each computing node where the parallel program is being executed. The agent is executed sharing the same memory space than the parallel program thus it is able to read program variables and modify its value. In addition, the mobile agent includes code for executing user-defined operations on each compute node. All these operations are performed asynchronously without interrupting/modifying the normal program execution. The agent is able to obtain information about the compute node (memory and CPU consumption, network use, etc.). All these information (related to each parallel program process and compute node) is collected in the agent private memory space, transferred along the agent itinerary compute nodes and finally stored and visualized in the front-end GUI. MASIPE allows the integration with any kind of C, C++ or Fortran parallel programs for distributed memory (MPI [6]) and shared memory (OpenMP [7]) architectures. Our tool is freely distributed. More information about MASIPE can be found in [8].

The structure of this paper is as follows. In Section 2 a description of the internal structure of each element of MASIPE is shown. Section 3 presents an example of its use for monitoring a parallel application that simulates the emission, transport and deposition of pollutants produced by a power plant. Next, Section 4 evaluates the impact of MASIPE on the program performance taking into account both memory and CPU requirements. Section 5 analyzes the related work and finally, Section 6 presents the main conclusions of this work.

2. MASIPE internal structure. Due to the massive expansion of Internet usage, distributed computation has become one of the most common kinds of programming paradigms. New alternatives have appeared, some examples of these are grid architectures, peer to peer systems, remote method invocation, web services, network services or mobile agent systems. MASIPE is a distributed mobile agent system that provides support for transferring and executing mobile agents in different compute nodes in an autonomous way. The communications between the compute nodes are implemented using the CORBA middleware. More specifically, we

*This work has been partially funded by project TIN2007-63092 of Spanish Ministry of Education and project CCG07-UC3M/TIC-3277 of Madrid State Government.

†Universidad Carlos III de Madrid, Computer Science Department, 28911 Leganés, Madrid, Spain. (dexposit@inf.uc3m.es, alejandro.miguel@gmail.com, fgcarbal@inf.uc3m.es, jcarrete@inf.uc3m.es).

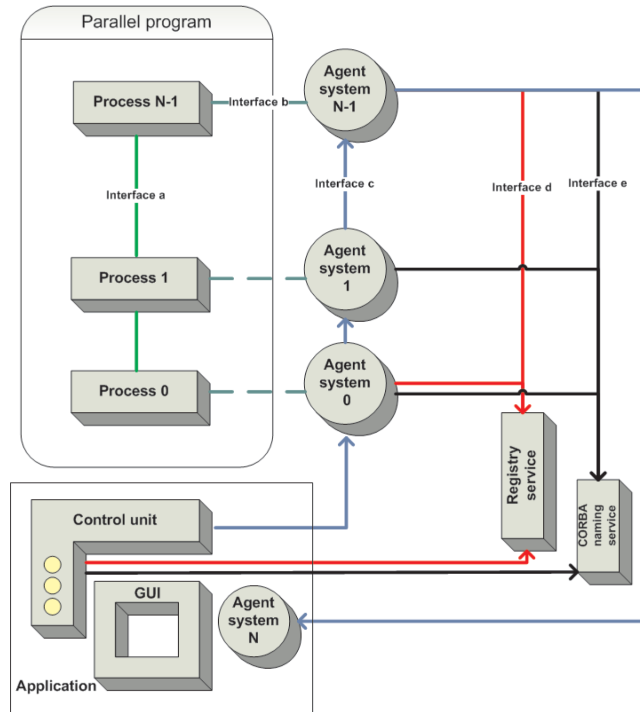


Fig. 2.1. *Diagram of MASIPE distributed and internal structure.*

have adopted the MASIF [9] specification proposed by OMG. This specification contains a description of the methods and interfaces of the agent-based architecture.

Figure 2.1 shows a diagram of MASIPE architecture as well as the relationships between its elements. MASIPE consists of three main components:

- **Agent System (AS).** This module receives the mobile agent, deserializes it, extracts its data and executes its associated code. When the agent execution concludes, AS collects and packs the agent code plus processed data, serializes it and sends it to the following compute node (where another AS is running). AS is attached to each parallel program process, sharing its memory space. This property allows reading or modifying (by means of the mobile agent) the data structure contents (variables or arrays) of the parallel program that is being monitored.
- **Register Service (RS).** This component has the task of registering AS and agents as well as providing its localization, that is, the IP address and port number of each element. Using this information, when a mobile agent arrives to an AS, the agent itinerary (stored with its data) is read in order to obtain the following AS where the given agent has to be transferred.
- **User Application (UA).** This module consists of three different elements: control unit, ending AS and GUI. The **control unit** creates the agent which is constituted by a header and a payload. The header contains the control data (including the agent itinerary). The payload contains the user-defined code (which performs user-defined operations) and its associated data. The control unit packs all this information (both code and data) and sends it to the first AS. Then, the agent is transferred along all the AS included in its itinerary. Finally, the **ending AS** receives the agent and unpacks all the collected data. These data are shown to the user by means of the **GUI** which includes components for graphically displaying the collected data. The User Application includes the following additional functionalities: it allows defining the system topology (agent itinerary); it performs agent tracking; it gives support for sending periodically (under a defined frequency) different types of agents; it receives, manages and displays the agent data and it includes policies for log generation.

All the communication operations described above (with the AS, RS and UA) are performed using pre-defined interfaces that are declared according the MASIF specification. Despite having this complex structure, the use of MASIPE is quite simple. Both RS and UA modules are independently executed as stand-alone

```
        First stage: add parameters
L1    call AS (0, 0, myrank)
L2    call AS (0, 1, it)
L3    call AS (0, 2, acum_time)
        Second Stage: start agent system
L4    call AS (1, 0, 0)
```

Fig. 2.2. *Example of a Agent System call in a Fortran program.*

programs. AS module is provided as a library that is linked with the parallel program subjected to be monitored. In addition, an *entry point* has to be defined in the parallel program. This point represents a call to the AS routine that starts executing the monitoring service.

Figure 2 shows an example of its use in a Fortran program. In the first stage, the user provides the program data structures that will be accessed by the agent. This task is performed in lines `L1`, `L2` and `L3` where calls to AS routine receive a pointer of three different program variables[1]. In this example, variables *rank*, *it* and *acum_time* are monitored by the agent. A zero value in the first argument indicates to the AS routine that has to store the given pointer value (third argument) in an internal data structure using the integer given by the second argument as an index. In the second stage, the Agent System is started calling the AS routine with a value in the first argument equal to one (line `L4`). Internally, a new thread is started and the code of the AS routine is executed concurrently with original process. Note that this process is automatically performed in each compute node where the parallel program is being executed.

3. Case of study: Monitoring STEM-II parallel application. In this section we show an overview of MASIPE functionalities when used in a real environment of parallel computing. More specifically, our proposal was used for monitoring STEM-II application.

STEM-II is a 3D grid-based model that simulates $SO_x/NO_x/RHC$ multiphase chemistry, long-range transport and dry plus wet acid deposition. This application is used for calculating the distribution of pollutants in the atmosphere from specified emission sources such as cities, power plans or forest fires under particular meteorological scenarios. The prediction of the atmospheric pollutants behaviors includes the simulation of a large set of phenomena, including diffusion, chemical transformations, advection, emission and deposition processes. This model was successfully used for the control of the emissions of pollutants produced by the Endesa power plant of As Pontes (Spain). In addition, STEM-II was chosen as case of study in the European CrossGrid project, proving its relevance for the scientific community from an industrial point of view as well as its suitability for the high performance computing.

STEM-II code structure is complex, consisting of more than 150 functions and 13.000 lines of Fortran code. STEM-II performs iterative simulation and has a multiple nested structure. The outmost loop is the temporal loop that controls the simulation time. Inner loops traverse the mesh dimensions computing for each cell (associated with 3D volume elements) the chemical processes and transport of the pollutants. This task is performed in parallel using the MPI standard library for implementing communications. A detailed description of the parallel program structure can be seen in [10].

For this scenario, two different agents were designed. The first type of agent (called `A-Type`) is used for tracking different STEM-II variables. More specifically, we have considered the following ones:

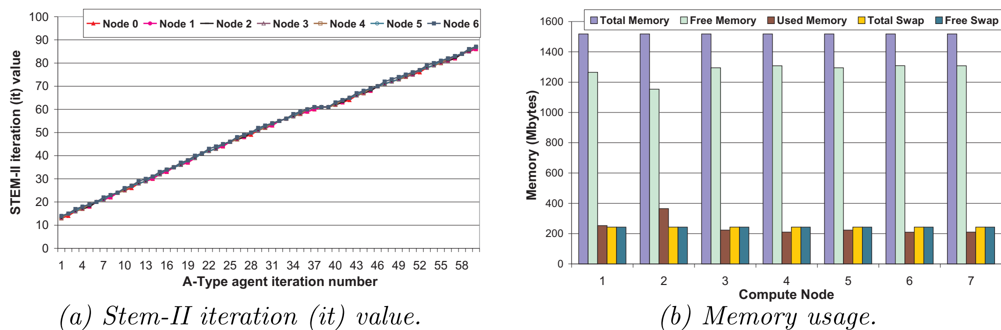
1. Rank identification (*myrank*). This variable is the MPI rank associated to each process. It is used for identifying each MPI processes during the communication operations.
2. Iteration (*it*). It corresponds to the iteration number of the outmost loop. Each loop iteration corresponds to a minute of the simulation time step.
3. Accumulated time (*acum_time*). It accumulates the execution time of the `rxn` routine. This routine performs the chemical simulation in each mesh node. `Rxn` is the most time-consuming routine of the program (70% of the complete program execution). Given that STEM-II is iterative, this routine is executed in each time step (outmost loop). In *acum_time* variable, the execution time of `rxn` is accumulated along different iterations.

[1]Note that in Fortran an argument variable of a routine is internally considered as a pointer. In case of programs written in C, standard pointers should be used instead.

An `A-Type` agent includes logic for reading, checking and modifying these parameters. In the case of *acum_time*, the mobile agent includes a threshold (with a value specified and initialized by the agent). When agent is executed it tests whether *acum_time* reaches this threshold. In case of being affirmative, the agent resets its value. Otherwise, its value is kept. In addition, the values of all these parameters are stored with agent data and further transferred to the UA for its visualization.

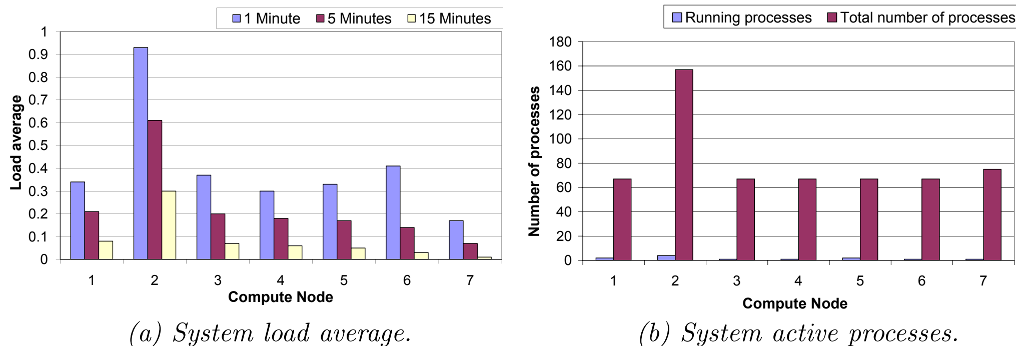
A second agent, called `B-Type`, was designed for collecting information about the compute nodes. Several features were collected; examples of them are the memory, CPU, disk-swap use and number of processes in execution.

In our experiments, we have executed STEM-II in a computer room of AMD Athlon(tm) XP 1700+ computers with 1.5 GB RAM memory, interconnected by a Fast Ethernet 100 Mbps network. Operative system is Linux Sarge 2.6.13. STEM-II was compiled with MPI-2 library LAM 7.1.1 and was executed using seven computing nodes. Regarding MASIPE, we used the Mico 2.3.12 CORBA distribution. We used an extra node for storing the CORBA name service and Register Service. Another extra node was used for executing the User Application component.



(a) Stem-II iteration (it) value. *(b) Memory usage.*

FIG. 3.1. *Examples of captured data using `A-Type` and `B-Type` agents.*



(a) System load average. *(b) System active processes.*

FIG. 3.2. *Examples of captured data using the `B-Type` agents.*

Taking advantage of MASIPE infrastructure, `A-Type` and `B-Type` agents were periodically sent for monitoring both program and computer parameters. Figure 3.1(a) shows the *it* value captured for the `A-Type` agent. X axis represents the agent execution number (agent iteration). That is, each `A-Type` agent has associated a new iteration number when completes the itinerary that has assigned. In each agent iteration, seven STEM-II *it* values are captured (one for each compute node). Note that these values are asynchronously read by the agent which monitors the parallel application without interfering with the program execution. Figure 3.1(b) shows the memory usage values captured by `B-Type` agent related to the memory use. In this case, the agent captures this information in each compute node.

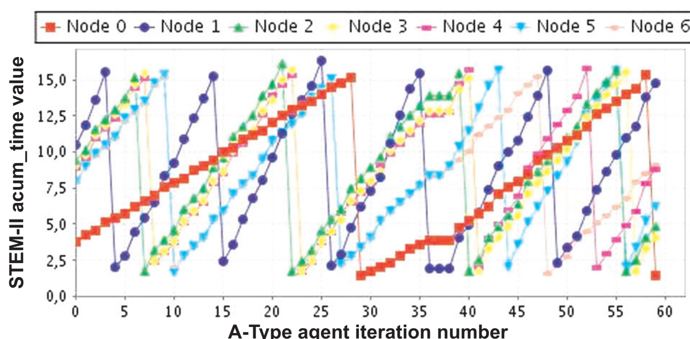


FIG. 3.3. *Example of data capture and visualization using MASIPE GUI.*

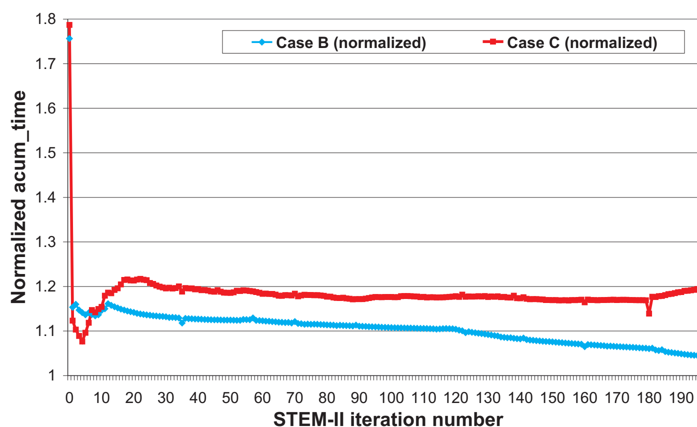


FIG. 4.1. *Evaluation of MASIPE overhead on STEM-II execution time.*

Figure 3.2 illustrates another example of captured values. Figure 3.2(a) shows the *load average* for last minute, 5 minutes and 15 minutes. This magnitude represents the average of processes in Linux's run queue marked running or uninterruptible during last minute. The load averages differs from CPU percentage in two significant ways:

1. Load average measures the trend in CPU utilization not only an instantaneous snapshot, as does percentage.
2. Load average includes all demands for the CPU not only how much was active at the time of measurement.

Note that in this test the compute node 2 has more computational load than the rest of the nodes. Figure 3.2(b) shows number of running processes and the total number of processes for each computational node. We can see that, again, compute node 2 has a higher number of processes than the rest of the nodes.

All values shown in Figures 3.1 and 3.2 were collected by the agent from each compute node and subsequently transferred to the User Application at the end of the agent itinerary. The User Application includes functionalities for storing in disk all these values, thus, they can subsequently be read, processed and displayed. In addition, the User Application includes a GUI for visualizing these data (as shown in these figures). Figure 3.3 shows an example of this visualization for the *acum_value* variable. Again, a `A-Type` agent was periodically sent, thus X axis represents the agent iteration number. But now, the *acum_value* variable is modified according a given threshold (with a value of 15). In the figure we can observe that the variable is reset when reaches the threshold. Note that each processor takes different time executing `rxn` routine thus *acum_value* increases at different rate for each node.

4. Performance evaluation. We have taken advantage of the computing environment described in the previous section for evaluating the impact of MASIPE in the parallel program performance. In order to evaluate all possible situations, three different scenarios where considered:

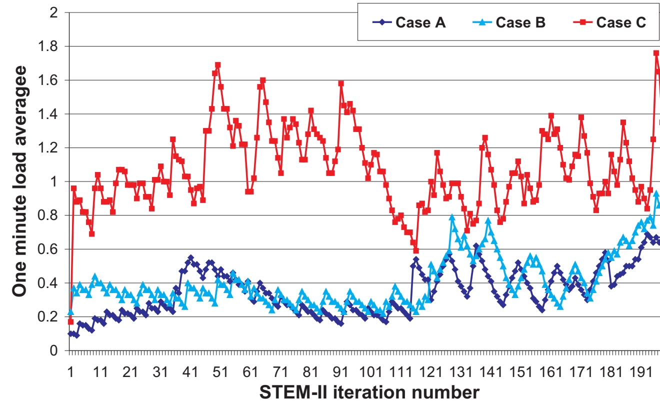


Fig. 4.2. *Evaluation of MASIPE overhead on load average.*

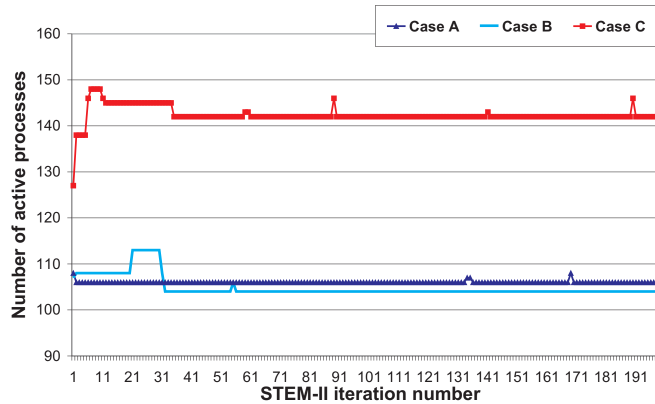


Fig. 4.3. *Evaluation of MASIPE overhead on number of active processes.*

1. **Case A:** STEM-II is executed as an stand-alone program. MASIPE tool is not used. This represents a reference scenario.
2. **Case B:** STEM-II is executed in combination with MASIPE but no agents are executed. In this scenario we evaluate the impact of the Agent System on STEM-II performance.
3. **Case C:** STEM-II is executed in combination with MASIPE. In addition, an agent is periodically sent with the maximum frequency that the system can achieve. In this scenario evaluates a full-operational MASIPE execution environment.

In all these cases, STEM-II is executed on seven compute nodes. In Case C we use a `A-Type` agent which performs an infinite round-circuit across the entire ASs. We modified STEM-II allowing collecting the performance measurements directly from this program in all the scenarios.

Figure 4.1 shows the impact of MASIPE on `rxn` subroutine. Results are normalized by Case A. We can see that the use of MASIPE without agents implies and increment around 10% on the `rxn` execution time. When agents are being periodically executed, this overhead reached up the 20% of `rxn` time. Note that we are considering only `rxn` routine because: Firstly, it is a compute-intensive routine with no E/S nor communications operations (we want to evaluate the CPU overhead) and secondly, it is the most consuming part of the program.

Figure 4.2 represents the one-minute load average for the three cases. In this figure we can see that the MASIPE load average overhead is minimal when no mobile agents are employed. Otherwise, the load average is incremented in one unit that means that are two CPU-consuming processes in execution[2].

Figure 4.3 shows the number of active process (including non CPU-demanding processes) for each one of the considered cases. Again, the use of MASIPE infrastructure increases this number in two processes. In contrast, when mobile agents are continuously used, the number of active process increases from 114 up to 142.

[2]Note that although the number of process in execution increases, the whole application execution time is not doubled as we can see in Figure 4.1.

This increment is not related to the MASIPE nor the mobile agent implementation but to the CORBA internal structure where multiple threads are automatically created.

The memory requirements of STEM-II (Case A) are 350 MB in each compute node. Starting the ASs (Case B) consumes 8 MB extra memory (an increment about 2%). When a mobile agent is fully operational (Case C) this extra memory rises up to 57 MB (16% increment respect to Case A). Taking into account the amount of installed memory in commodity computers, the memory requirements of MASIPE for this case of study are reduced.

5. Related work. There are many cluster administration tools. One well known example is Rocks [1]. Rocks is a management tool that makes complete Operating System installation and its further administration and monitoring. Other relevant schemes for system monitoring are Ka-admin project [2], Vampir [3] and Paraver [4]. However, all of them do not include functionalities for monitoring parallel applications accessing to their memory space.

Example of agent systems are Jade [11] and Mobile-C [12]. The agent mobility in Jade is achieved through Java object serialization, but Java implementation limits the integration with C/C++/Fortran codes. Mobile-C uses ACL messages for mobile agent migration which is an alternative to our implementation based on CORBA. In [13] an instrumentation library for message passing parallel applications, called GRM, is presented. GRM collects trace data from parallel applications using a similar way than our scheme. However, there are important differences in the tool architecture: in GRM all the ASs send the data to the User Application introducing contention risks in this point. The integration with Mercury Monitor presented in the paper does not solve this drawback. In contrast, we use a mobile agent program paradigm that reduces these risks given that the agent traverses different ASs. The agent itinerary in user-defined and can be dynamically changed (adding decision logic to the agent). In addition, with MASIPE it is possible to introduce new designed agents without changing (even restarting) the tool. This strength provides MASIPE a broad range of uses. Finally, CORBA allows using our tool on heterogeneous platforms.

6. Conclusions. MASIPE is a distributed application that gives support to user-defined mobile agents, including functionalities for creating and transferring the mobile agent along the compute nodes of its itinerary. In each node, the mobile agent can access to the node information as well as the parallel program memory space. In addition, MASIPE includes functionalities for managing the agent data.

MASIPE allows the integration of user-defined code (included in the mobile agent) with a generic parallel/sequential program. This process only requires of a single functional call to ASs (represented as a dynamic library). The mobile agent can not only read the program data but also modify it. This introduces a vast number of possibilities of dynamic interaction between the mobile agent and the program: check pointing (the mobile agent collects and stores the critical data of the program), convergence checking (the mobile agent evaluates the numerical convergence of iterative me-thods, even introduces corrections), evaluation of the program and system performance, etc. These interactions are performed asynchronously, without interfering with the monitored program.

The tool design was performed based on MICO Corba and Java IDL following the MASIF specification, which strengths its capacity of use on heterogeneous platforms (including compute rooms, clusters and grid environments). The installation requirements are minimal, being necessary just a single dynamic library (AS component) in each compute node. No administrator grants are needed for its use.

MASIPE was applied with success to monitor STEM-II parallel application. Several tests were made including reads and writes in the application memory space as well as capturing information of each compute node. Experimental results show that MASIPE overhead is reduced both in terms of CPU and memory consumption.

As future work we plan to develop more MASIPE functionalities like new database functionalities for managing the data collected by the mobile agents (specially in checkpoint operation), improving the GUI for dynamically visualizing the distributed data (obtaining an snapshot of the simulation), introducing new commands for controlling the program behavior (I/O operations, start and resuming the program) and its application to grid platforms.

REFERENCES

[1] F. D. Sacerdoti, S. Chandra and K. Bhatia, *Grid systems deployment & management using Rocks,* in CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing. (2004) pp. 337–345.

[2] P. Augerat, C. Martin and B. Stein, *Scalable Monitoring and Configuration Tools for Grids and Clusters,* in 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing. (2002) pp. 147–153.
[3] *Vampir*, in `http://www.vampir-ng.de`
[4] *Paraver: the flexible analysis tool*, in `http://www.cepba.upc.edu/paraver`
[5] David E. Singh, Alejandro Miguel, Félix García, Jesús Carretero, *MASIPE: A tool based on mobile agents for monitoring parallel environments,* Third Workshop on Large Scale Computations on Grids in conjunction with the Seventh International Conference on Parallel Processing and Applied Mathematics 2007. Lecture Notes in Computer Science. **4967**. (2008).
[6] William Gropp and Ewing Lusk and Anthony Skjellum, *USING MPI Portable Parallel Programming with the essage-Passing Interface,* in The MIT Press. (2004).
[7] R. Chandra, *Parallel programming in OpenMP,* in Morgan Kaufmann Publishers. (2001).
[8] *Mobile Agent Systems Integration into Parallel Environments (MASIPE)*, in `http://www.arcos.inf.uc3m.es/~masipe`
[9] *OMG Mobile Agent Facility Specification* in `http://www.omg.org/technology/documents/formal/mobile_agent_facility.htm` (2007)
[10] M. J. Martn, D. E. Singh, J. Carlos Mourio, F. F. Rivera, R. Doallo and J. D. Bruguera, *High performance air pollution modeling for a power plant environment,* in Parallel Computing. **29**, (2003) pp. 11–12.
[11] F. Bellifemine, A. Poggi and G. Rimassa, *Developing Multi-agent Systems with JADE,* in Lecture Notes in Computer Science. **1986**, (2001) pp. 42–47.
[12] B. Chen, H. H. Cheng and J. Palen, Mobile-C *A mobile agent platform for mobile C/C++ agents,* in Software. Practice and Experience **36**, (2006) pp. 1711–1733.
[13] N. Podhorszki, Z. Balaton and G. Gombás, *Monitoring Message-Passing Parallel Applications in the Grid with GRM and Mercury Monitor,* in European Across Grids Conference. (2004) pp. 179–81.

Edited by: Dana Petcu, Marcin Paprzycki
Received: May 5, 2008
Accepted: May 20, 2008