Grid-Enabled Distributed Simulation – A State Machine Based Approach

Cosmin Marian Poteras, M.Sc., PhD. Student, and Mihai L. Mocanu, PhD., Professor, *University of Craiova*, Romania

Abstract — **We introduce in this paper a new approach for grid-enabled distributed frameworks designed for distributed simulation and computational steering, which can easily be used for general purpose processing. Previous research focused on the development of a set of frameworks and platforms for distributed simulation and computational steering. Most of these frameworks were dedicated to certain projects or they treated only a part of the issues that needed to be dealt with in this field. Our approach, which uses the state machine concept, is more general and scalable, the main benefits of state machine encapsulation being safety and flexible task migration. An image processing case study will be presented to illustrate the feasibility of our approach.**

Keywords — **computational steering, distributed framework, grid computing, simulation, state machine.**

I. Introduction

MODELING and simulation are very important phases in any modern research. In contrast with the study of a real system, whose advantage is the accuracy of the evaluation, but who might become destructive, dangerous, and expensive, the study of a model is easier, safer and cheaper. Due to their impressive development, grid environments have become the most appropriate platforms for complex simulations. Simulation refers to the numerical evaluation of a model. When dealing with a complex system whose analytical solution is hard to be determined, simulating the system's behavior on a model might be of great help. By the late 80s the simulation had been considered a tedious and time consuming process, due to the lack of interactivity with the ongoing simulation process. The researcher had to exhaustively execute the simulation for all input data sets and he could only analyze data as a post-simulation phase, even if in some cases the simulation process reveals useless results from the beginning. Therefore the need of interactivity became obvious, and researchers concentrated on developing simulation frameworks with steering capabilities, so that the ongoing simulation could be guided, observing immediately the impact.

The most relevant frameworks for distributed simulation and computational steering have proven to be: RealityGrid, CUMULVS and CSE.

Mr. Cosmin Poteras and Dr. Mihai Mocanu are with the Department of Software Engineering at the University of Craiova, Romania. Contact e-mail: {cpoteras, mmocanu}@software.ucv.ro

Additional financial support for this work has been provided to Cosmin Poteras by AMPOSDRU grant 109/25.09.2008.

RealityGrid [1], [2] project consists of an API library divided in two main modules: one for developing steering features and another for building the client application. RealityGrid-based applications' architecture has to be adapted to the library's requirements. The steering capabilities are reached by using a check pointing technique that assumes the handling of steering events at certain points in the application's runtime.

CUMULVS (Collaborative User Migration, User Library for Visualization and Steering) [3] is a software platform designed for the development of collaborative environments, developed at Oak Ridge National Laboratory. The collaborative, on-line and interactive visualization, the remote computational steering of distributed applications in distributed environments, the automatic recovery of the virtual environment in case of a host or network failure, the powerful fault-tolerance mechanism based on tasks migration and check pointing capabilities, make CUMULVS a very powerful framework for distributed and interactive simulations.

CSE (Computational Steering Environment) [4], [5] has been developed at the Center for Mathematics and Computer Science, in Amsterdam. Its architecture is built around a module called Data Manager which is responsible for managing the system's parameters, and the notifications related to the parameters' state. The data manager communicates with simulation processes, called satellites, which can reside on different hosts and which carry out the computation. To reduce traffic the data manager can be replicated on multiple hosts.

The rest of this paper is organized as follows. In section 2 the architecture of the proposed framework is introduced. Section 3 describes a sample application built for evaluating the performance brought by the framework. The conclusions, as well as our future research issues are depicted in section 4.

II. State Machines In Grid Environments

In certain fields (like medicine), dedicated software requires high performance environments while it is incompatible with errors and instability (imagine a software that assists a surgery). To improve the reliability, one has to make sure that at any moment the software is consistent. This can be achieved by analyzing all states prior to running the software and making sure its reaction in each state is appropriate, requirement which led to the idea of representing simulation processes as finite state machines.

The approach has the following advantages: code safety and robustness, traceability, avoid erroneous states and inconsistencies, simplified approach for complex tasks, easier to test, complete documentation. Besides safety, one can benefit of the state machine's encapsulation when dealing with tasks migration. State machines are easier to pack, transfer, unpack and resume. The ability of tasks to migrate opens a new path for defining automatic and dynamic load balancing algorithms by considering a set of performance and workload coefficients for hosts (available CPU power, network bandwidth, host's reliability, physical resources, etc.).

Although load balancing algorithms proved to be very efficient in distributed environments, in some cases it would be also useful to allow users with technical skills to steer the tasks migration, bringing the computational steering to a more technical level.

The class library proposed in this paper is part of a more general distributed simulation environment whose architecture (illustrated in Fig. 1) consists of five main modules: Simulation Module, Control and Communication Module, Visualization Module, Shared Memory Module and Client Application.

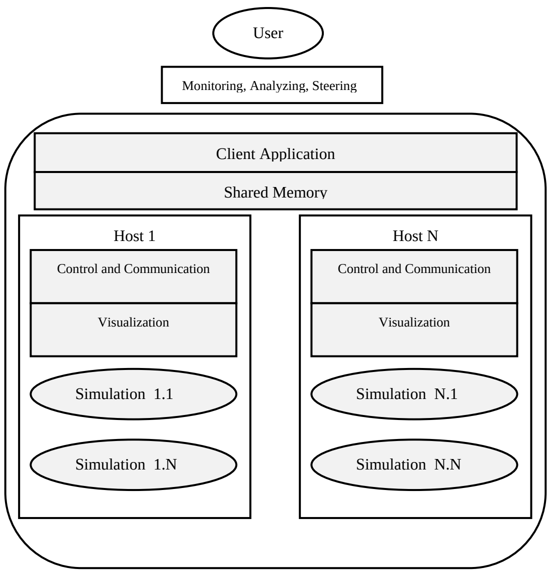

Fig. 1. The structure of the proposed distributed system.

The simulation module is the one that performs all the parallel computation. Simulation processes are represented as state machines, and there might be many processes running on the same host. The shared memory module is an abstract module which can take either the form of a distributed shared memory or a centralized shared memory held by a server host. The control and communication module handles two types of tasks: communication tasks (acquiring the input data and forwarding it to the simulation processes, acquiring the output data from the simulation processes and forwarding it to the visualization process, synchronizing the access to the shared memory) and control tasks (monitoring the workload and available resource of each host, deciding and managing the migration of tasks - stop/send or receive/start the simulation processes). Each host will run only one instance of this module. The visualization module acquires the simulation's output, filters it, and translates it into the proper format for visualization. The client application is the one that initializes, monitors, controls (steers) and analyzes the simulation.

Before proceeding with the library's architecture, a theoretical model of state machine has to be considered. A state machine is a quintuple $M = (\Sigma, S, s_0, \delta, F)$ where Σ is the set of input parameters (input alphabet, finite, non empty), S is the set of states, s_0 is the initial state, δ is the states transition functio n $\delta : \Sigma \times S \to S$ and F is the set of final states. The architecture has been built based on the above model and on the idea of separating the machine's code from machine's data.

The library consists of a set of abstract classes and interfaces, enabling the user to define the custom behavior of the state machines, as well as an engine for machines' migration (from one host to another) and runtime machines' management.

A. *State Machine's Workflow*

Fig. 2 illustrates the library's class diagram. The core class is the StateMachine class which is abstract and exposes the method performComputation, responsible with the computations required by each state. This method will handle all the application's logic.

The StateMachine's code and data are separated by using a StateMachineData class, where all data necessary for running the machine is encapsulated: the parameters (IParameters), the states transition matrix (TransitionTable), the current state (currentState), the identifier of the machine (ID) unique in the entire environment, the exact type of the machine (type) representing the actual class of the machine object – needed when machine reaches a new host and needs to be instantiated, and the set of final states.

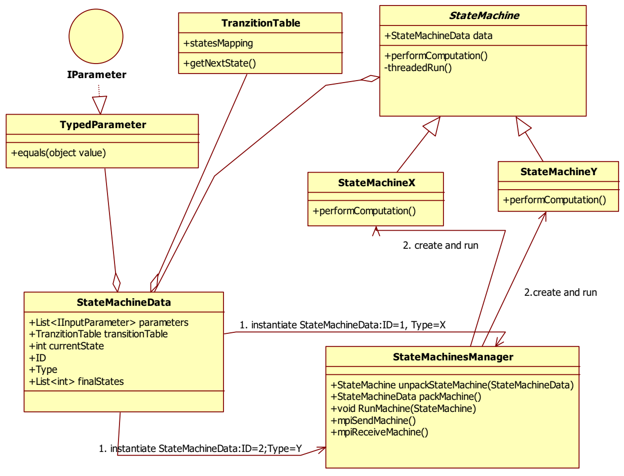

Fig. 2. The Library's architecture and workflow.

The TransitionTable class is represented as a mapping between pairs <parameters, state> and future states. The mapping might only use a subset of the machine's parameters. The method getNextState is responsible for finding the machine's next state based on the current values of its parameters and state. As the input parameters have to be handled in the same way, no matter their actual type, they are represented by the interface IParameter which exposes the method equals, enabling parameters' matching.

B. *State Machine's Migration*

Each host process has to be able to run its own set of state machines, as well as to allow their migration. To achieve these features, the StateMachinesManager has been designed. On each host, an instance of the StateMachinesManager will run. Its main role is to pack/unpack the machines, run then and send/receive them.

Migrating a state machine consists of the following steps:

1. Packing – extracting the StateMachineData object
2. Sending the StateMachineData object to the destination host via MPI. In MPI .Net this is a very simple task, due to the serialization feature that .Net Framework offers.
3. Receiving the StateMachineData object by the destination host.
4. Unpacking the StateMachine object – this requires dynamically creating an instance of the actual StateMachine derived type, employing the StateMachine.type field.
5. Running the machine inside the destination host's state machines manager.

III. A SAMPLE APPLICATION

Due to the architecture presented in section 2, implementing a new application consists of the following steps:

1. Implementation of the IParameter interface for each of the machine's parameters types.
2. A new derived StateMachine class has to be defined for each type of state machine; the performComputation method will contain the entire machine's logic.
3. An instance of the TransitionTable has to be passed to every state machine.
4. A StateMachine manager has to be instantiated on every host. The load balancing algorithm will reside in this class.

A sample image processing application has been implemented on top of the framework. The only purpose of this application is to give an overview about the potential gain in terms of computational performance offered by the proposed framework and to prove the general purpose capability of the framework. The application does not relate to any simulation specific aspects. All the application does, is to load image data, apply filters on it, output the resulted image and measure the time needed for processing. Images will be loaded, split in pieces and passed to state machines (one piece for each state machine). In each state, the machines will process a piece of the input image. The load balancing algorithm used, decides whether to send machines to another host or not based on the CPU and memory availability. Each state machines manager process will send information about the system's load to all others, and each of them will decide how many machines will be sent and what will be the destination host.

The development environment chosen for this application consists of the Microsoft .NET Framework combined with MPI .NET. The choice has been based on the powerful serialization features of .NET Framework which bind perfectly together with MPI .NET, a well known, stable and efficient platform for distributed applications.

As loading data from different types of storage drives across the network might introduce important delays and affect the processing time, special techniques need to be considered to speed up the data flow. As the purpose of the application is only related to the computational power of the framework, all the experimental results presented in this section will assume that all the necessary data has been loaded in memory prior to starting the experiments.

The application has been deployed and evaluated in both a Myrinet and Ethernet networks using a JPEG image (1.26 Mb, 2048x1536, 180dpi, 24bpp).

In the following table there are presented the results measured for the framework against the results measured when running the same processing without using the framework, on one host, in parallel and sequential modes.

TABLE 1: EXPERIMENTAL RESULTS.

	No. of hosts	***No. of instances per host***	***Time for Gaussian Filter (s)***
Sequential*	1	1	34.406
Multi-threaded*	1	4	15.804
Ethernet	4	4	11.339
Myrinet	4	4	9.282

* Not using the framework

Besides the image processing application built on top of our framework, there have also been developed two more simple applications: the former is a single-threaded application that runs the Gaussian filter on one thread, and the latter is a multi-threaded application which splits the image in as many parts as processing threads and runs the Gaussian filtering on every part.

The Myrinet network used for this experiment consist of 4 hosts with Intel Core 2 Duo E5200 processors and 1GB of memory. The Ethernet network consist of 4 hosts with Intel Core 2 Quad processors and 4 GB of memory.

IV. CONCLUSIONS AND FUTURE WORK

The class library introduced in this paper is part of a more complex project whose main goal is to offer an enhanced distributed environment for simulation and computational steering.

The library proved to be scalable, flexible and optimizes considerably the development process by handling the tasks migration.

The technologies chosen for the implementation are the .NET Framework and MPI .Net. The choice was based on the promising cross-platform character of .NET Framework as well as its serialization feature which binds perfectly with MPI .Net.

In order to evaluate and analyze the framework, a Myrinet cluster as well as an Ethernet network have been used.

Our future research will be focused on extending the framework by designing optimal automatic load balancing algorithms for both data flow and hardware resources use, new simulation and steering capabilities, as well as developing optimal visualization techniques [6].

REFERENCES

[1] S. Jha, S. Pickles, and A. Porter, *A Computational Steering API for Scientific Grid Applications: Design, Implementation and Lessons.* In *Workshop on Grid Application Programming Interfaces*, Brussels, Belgium, Sept. 2004

[2] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning and A. R. Porter, *Computational Steering in RealityGrid*, Proceedings of the UK e-Science All Hands Meeting, September 2-4, 2003

[3] J. A. Kohl and P. M. Papadopoulos, *Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS*. In *2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, Welches, OR, Aug. 1998

[4] J.J. van Wijk and R. van Liere, *An environment for computational steering*. In G.M. Nielson, H. M¨uller, and H. Hagen, editors, *Scientific Visualization: Overviews, Methodologies, and Techniques*, pages 89–110. Computer Society Press, 19997

[5] R. van Liere, J.D. Mulder, and J.J. van Wijk. *Computational steering*. *Future Generation Computer Systems*, 12(5):441–450, April 19997

[6] Poteras Cosmin Marian, Gosa Mihai, Mocanu Mihai. *Parallel Visualization on GPU with CUDA*, Proceedings of the 4th Annual South-East European Doctoral Student Conference. pg. 480-487. July 2009.