A State Machine-Based Parallel Paradigm Applied in the Design of a Visualization and Steering Framework

Cosmin M. Poteras and Mihai L. Mocanu

Abstract—This paper introduces a new approach, based on state machines, for distributed frameworks, that is able to support both distributed simulation and computational steering. Existing frameworks are in general dedicated to certain projects or they only take care of a part of the issues related to this field. Our approach is scalable, flexible, supports load balancing and tasks migration while providing a key feature for any complex system: safety.

Keywords—computational steering, state machines, distributed simulation

I. INTRODUCTION

RUNNING complex applications in today’s world is more and more a matter of integration of efficient infrastructures and good computational techniques. Cluster and grid simulation applications that employ parallel computing techniques (i.e. MPI, OPENMP) to simulate real processes are just a common example. Modeling and simulation have become key phases for a wide spectrum of applications in 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. Modeling, as a general term, denotes the process that offers an abstract representation of a system, which allows, in turn, through its study, the formulation of valid conclusions on the real system. Simulation generally 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 could be the only solution left.

The outcome of a simulation may be analyzed in a separate post-processing step (for instance by viewing the results in a separate visualization application), and, based on intermediate results, a decision can be made to change simulation parameters for another computational period.

In order to increase the efficiency, new techniques for live visualization and steering have emerged allowing simulation and visualization to be performed simultaneously. If online visualization refers to the ability to immediately observe the processing steps during the simulation, this in turn allows for computational steering to influence the computation of the simulation during runtime on a cluster, grid and even on a supercomputer. They are meant together to dynamically steer the parameters of a parallel simulation, increasing not only the interactivity but also the efficiency of the overall process.

Visualization is the process of exploring, transforming and rendering of data through images, with the goal to offer a thorough and deep understanding of data. It is a complex field of study in our days, including elements of computer graphics, digital image processing, computational geometry, numerical analysis, statistics, and studies on human perception.

Computational steering is a process of manual intervention on an autonomous computational system, with the goal to analyze and modify outputs. It is a very common technique in numerical evaluation, used to guide a computational process towards regions of interest. Apart from this pure applicative perspective, computational steering can be examined from a broader technical perspective; for instance, we may consider the modification of memory amount available for a process, with the goal to observe and influence the effects over the execution time. This paper deals with the concept especially in the latter, broader sense. The taxonomy of the concept also includes: program steering, which has been defined as the capability to control the execution of resource-intensive, long-running programs (this may imply modifications of program state, starting and stalling program execution, etc.), data steering (which implies the management of data output, alteration of resource allocations etc.), and dynamic steering (which requires the user to monitor program or system state and have the ability to make changes, through “add-ons” routine calls or data structures interaction in the code).

Interactive simulation combined with visualization has undergone a major development and it is now widely used. Up to the late 80s, simulation had been considered a tedious and time consuming process, mainly 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. When the need of interactivity became obvious, research also concentrated on developing simulation frameworks with visualization and steering capabilities, so that an ongoing simulation could be immediately observed and guided. The development of distributed simulation and steering frameworks, able to support run-time adjustments and live visualization, has not been an easy task.

Extensive surveys of research in this area were carried out in over the last two decades [1, 2], however not many of the
projects led to practical tools. Some of the most relevant frameworks for distributed simulation and computational steering, for the scope of this paper, may be considered: COVS[1], RealityGrid, CUMULVS and CSE.

COVS[3] (Collaborative Online Visualization and Communication) is a framework that encapsulates common visualization frameworks (VTK, AVS/Express), steering technologies (VISIT, gViz, ICENI) as well as communication libraries (VISIT, PV3) that carry out the data transportation and steering commands. This multi-framework integration allows COVS to run simulations independently from visualization and communication tasks.

RealityGrid [4, 5] is an API library consisting mainly from two modules. The former is responsible for offering steering capabilities and the latter provides tools for dedicated client applications. RealityGrid uses check-pointing techniques for supporting steering commands.

CUMULVS (Collaborative User Migration, User Library for Visualization and Steering) [6, 7] has been developed at Oak Ridge National Laboratory and has been designed for the development of collaborative on-line and interactive simulation and visualization. The power of this platform consists in the advanced recovery techniques, the tasks migration support and check-pointing.

CSE (Computational Steering Environment) [8, 9] has been developed at the Center for Mathematics and Computer Science, in Amsterdam. It uses a centralized architecture around a replicated Data Manager that is able to carry out steering commands and coordinate the simulation tasks.

The Data Manager from CSE leads us to an important problem in the analysis of these efforts: data availability. The computations may be dramatically slowed down by the acquiring of data. Dataflow processing is at the same time the most appropriate model of programming and a crucial factor for achieving the desired performance. Existing systems like BitTorrent and Apache Hadoop Distributed File System implement a parallel dataflow style of programming which provide the data required by a distributed application’s processes in the most efficient way.

The BitTorrent Protocol [10] establishes peer-to-peer data transfer connections between a group of hosts, allowing them to download and upload data inside the group simultaneously. The torrents systems that implement BitTorrent protocol use a central tracker that is able to provide information about peers holding the data of interest. Once this data reaches the client application, it tries to connect to all peers and retrieve the data of interest. However, it is up to the client to establish the upload and download priorities. Torrents systems might be a good choice for distributed environments, especially for those based on slower networks. However, the main disadvantages of torrent systems are related to the centralized nature of the torrents tracker as well as leaving the entire transfer algorithms and priorities up to the client application which might cause important delays if the transfers trading algorithm chooses to serve a peer that might have a lower priority at the application level. The centralized nature of the tracker concentrates the reliability around the tracker; if the tracker goes down, the entire system becomes not functional. Torrents are mainly systems that transfer files in distributed environments in raw format without any logical partitioning of the data. Such logical partitioning might often prove to be very important. For example if an imaging application needs a certain rectangle of an image it would have to download the entire file and then extract the rectangle by itself instead of just downloading the rectangular area and avoid transferring unnecessary parts of the file.

The Hadoop Distributed File System has been designed as part of Apache Hadoop [11] distributed systems framework. Hadoop has been built upon the Google’s Map-Reduce architecture as well as HDFS file system. HDFS proved to be scalable, and portable. It uses a TCP/IP layer for internal communication and RPC for client requests. The HDFS has been designed to handle very large files that are sent across hosts in chunks. Data nodes can cooperate with each other in order to provide data balancing and replication. The file system depends closely on a central node, the name node whose main task is to manage information related to directory namespace. HDFS offers a very important feature for computational load balancing, namely it can provide data location information allowing the application to migrate the processing tasks towards data, than transferring data towards processing task over the network [4]. The main drawback of HDFS seems to be the centralized architecture built around the name node. Failure of the name node implies failure of the entire system. There are still available techniques for replication and recovering of the name node, but this might cause unacceptable delays in a high performance application.

Due to the well known diversity and complexity of distributed models, choosing the appropriate design for a system like ours is not an easy task. Besides of the usual requirements imposed to a distributed system, like scalability, flexibility, extensibility, portability, we added support for load balancing and tasks migration, and safety features. For this we concluded to a design that merges together parallel mapping of tasks in the form of state machines, able to be deployed in a robust way over a network, and parallel dataflow handling, separated into a standalone module whose main role is to acquire, store and provide the data required by the application’s processes in the most efficient way. We will describe in this paper only the design of the first module.

The rest of the paper is organized as follows. Section II describes the state machines framework design and some implementation details. In Section III we review two simple applications that have been implemented on top of the framework; their main role was to validate the design, offering an overview on the potential of framework’s architecture. This section also gives some preliminary experimental results. Section V concludes the paper and discusses some of the many possibilities we have for further developments, due to the extensibility feature we imposed for design.
II. STATE-MACHINES FRAMEWORK DESIGN AND IMPLEMENTATION

It is very common that many domains impose very strict requirements for software. For example, medicine requires very high safety standards as well as high performance environments due to the incompatibility of this field with errors and instability. Imagine the dramatic effects of an error occurred in a software application that assists a surgery. That could become fatal. To improve the reliability and safety, one has to make sure that at any moment the software is in a consistent state. A good practice would be to analyze all possible states prior to the system development and by ensuring the system’s reaction is appropriate in any state. All these constraints lead us to the idea of representing tasks as finite state machines.

State machines can provide code safety, robustness, traceability, excludes erroneous states and inconsistencies while providing a simple and well structured “package” for representing complex tasks. Being represented as “packages”, tasks are encapsulated and can easily migrate in distributed environments. Tasks migration together with live monitoring of the distributed environment reveals new possibilities for defining dynamic load balancing algorithms.

We propose in this paper a new design model for distributed simulation environments whose architecture is illustrated in fig.1. The model has been implemented as a class library that reduces considerably the applications development time. Our model consists of five main modules: Simulation Module, Control and Communication Module, Visualization Module, Shared Memory Module and Client Application.

The processing is being performed by the simulation processes. They are represented as state machines, and there can be run as many processes as each host can handle efficiently. The shared memory module can comply either to a distributed form or a centralized one. Its main goal is to store the system’s parameters which usually realize the computational steering. The control and communication module handles data flow as well as monitoring and migration jobs. It is responsible for acquiring input data, forwarding output data to the visualization filters, synchronizing access to the shared memory while monitoring the system’s resources and loads and realizing machines’ migration whenever necessary. The control and communication module is able to rise the computational steering to a new level by allowing the user to manually specify simulation processes migration. There will be only one instance of the control and communication module on each host. The visualization module is responsible for translating simulation’s output which usually is in a raw format into a more appropriate format for visualization. The client application initializes, monitors, controls (steers) and analyzes the simulation.

The architecture is based on the theoretical model of a state machine: a state machine is a quintuple \( M = (\Sigma, S, s_0, \delta, F) \)

where \( \Sigma \) is the set of input parameters (input alphabet, finite, non empty), \( S \) is the set of states, \( s_0 \) is the initial state, \( \delta \) is the states transition function \( \delta:\Sigma \times S \rightarrow S \) and \( F \) is the set of final states. The architecture has ensures the separation between machine code and machine data

The library consists of a set of abstract classes and interfaces that allow the developer to define the machine’s algorithm by extending/implementing the proper methods. The library’s engine automatically manages the state machines and their migration.

![Fig. 1. The structure of the proposed distributed system.](image)

The main class of the platform is the StateMachine class. It is an abstract class which serves as base class for every type of state machine required by the application (StateMachineX, StateMachineY). It handles the states succession and computations by employing the performComputation method together with the states transition table. The performComputation method will be overridden by the derived types and it will hold all custom algorithms specific to each
In this section we’ll consider first a simple application that multiplies two matrices. Matrix multiplication is being performed by multiplying elements from the first matrix’s lines with their corresponding elements in the second matrix and by computing the sum of all these products.

In our application there will be started as many machines as lines in the first matrix. Each machine will compute a pair line-column. The processing will be performed in two steps (states). The first state will process the products while the second will process the sum of products resulting an element from the result matrix. The states transition will not be influenced on the parameters (line, columns) values, instead it will jump from first state to the second which is also a final state.

The two matrixes will be split into as many arrays as lines or columns respectively, they have. Thus, we need to implement the parameter type IntArrayParameter (implementing IParameter) which holds a line or a column of the matrixes as an array of integers. Each state machine will receive two parameters as input: a line from the first matrix and a column from the second.

The next step implies creating the class MatrixMultiplyStateMachine which derives from the StateMachine class and overrides the method performComputation. This method will be called once for each of the two states.

Class StateMachineData will be instantiated once for each state machine and will hold the two IntArrayParameter instances mentioned above (the line and the column), the initial state (state 0), the final state (state 2), and the transition table as can be observed in table 1.

### Table 1: States Transition Table

<table>
<thead>
<tr>
<th>Parameter 1 (Matrix 1 line)</th>
<th>Parameter 2 (Matrix 2 column)</th>
<th>Current State</th>
<th>Future State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any value</td>
<td>Any value</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Any value</td>
<td>Any value</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

As a final step the StateMachineManager will be instantiated and run on each host.

In order to continue towards the evaluation of the system, but also for the validation of design, another simple application that has been implemented on top of our framework will be presented in the remainder of this section. It should be noted that the results presented here are only preliminary results from a run in a small distributed high performance environment, for offering an overview on the potential of framework’s architecture. The processing tasks of the application refer to loading images and applying imaging filters on them. To perform the task, the images are split into small rectangular pieces, each piece being passed to a state machine. In each state the machine will perform a part of the filter’s specific computation. The load balancing algorithm will be based on the available CPU power as well as the

---

StateMachine class starts computations by invoking the method passing as parameters the initial state, performs computations associated with this state and retrieves the output. The states transition table is being checked for the next state and the process continues in the same manner.

Data is being separated from code by using StateMachineData objects. StateMachineData holds all relevant information about the machine: parameters, current state, transition table, machine identifier – unique in the entire environment, the machine type (StateMachineX, StateMachineY), final states, etc. All these can be extended by deriving the StateMachineData class.

The transition table (TransitionTable) it’s represented as a mapping between pairs <parameters, state> and future state. The transition is performed by method getNextState which retrieves the next state based on the current state and the output values of the parameters from the current state.

For flexibility reasons the parameters have been interfaced by the IParameter interface leaving its implementation up to the developer. IParameter offers getter and setter methods as well as parameters matching methods.

The state machines’ management is ensured by the “brain” class, which is StateMachinesManager. Its role is to manage all the machines running on a host. It is able to monitor the system, to ensure data availability, to create, run and migrate machines to and from other hosts. The most important tasks performed by the StateMachinesManager are related to tasks migration and load balancing. These tasks are performed by the following methods: packMachine() – prepares the machine for migration, unpackMachine() – prepares machine for resuming the processing on the new host, mpiSendMachine() – sends the machine to other host, and mpiReceiveMachine() – receives the machine from another host, and RunMachine() – which resumes the processing. Each host in the distributed environment will run one instance of the state machines manager.

Considering the above implementation details we can enumerate the steps needed for implementing distributed simulation applications on top of the framework.

- **Defining the parameters of the system (implementing IParameter)**
- **Defining all types of machines needed.** For each type, a new derived class will be created inheriting the class StateMachines. The method performComputation will hold the processing algorithms.
- **TransitionTable class will be instantiated and populated with mappings of type <<parameter, current state>, future state>**
- **StateMachinesManager will be instantiated and run on each host.**

For a better understanding of the system’s architecture, an implementation example will be presented in the next section. It will not be a complex distributed simulation, due to the lack of space, but a very simple application in which the regular parallelism is easy to be discovered.

### III. Experimental Results

The state machines' management is ensured by the “brain” class, which is StateMachinesManager. Its role is to manage all the machines running on a host. It is able to monitor the system, to ensure data availability, to create, run and migrate machines to and from other hosts. The most important tasks performed by the StateMachinesManager are related to tasks migration and load balancing. These tasks are performed by the following methods: packMachine() – prepares the machine for migration, unpackMachine() – prepares machine for resuming the processing on the new host, mpiSendMachine() – sends the machine to other host, and mpiReceiveMachine() – receives the machine from another host, and RunMachine() – which resumes the processing. Each host in the distributed environment will run one instance of the state machines manager.

Considering the above implementation details we can enumerate the steps needed for implementing distributed simulation applications on top of the framework.

- **Defining the parameters of the system (implementing IParameter)**
- **Defining all types of machines needed.** For each type, a new derived class will be created inheriting the class StateMachines. The method performComputation will hold the processing algorithms.
- **TransitionTable class will be instantiated and populated with mappings of type <<parameter, current state>, future state>**
- **StateMachinesManager will be instantiated and run on each host.**

For a better understanding of the system’s architecture, an implementation example will be presented in the next section. It will not be a complex distributed simulation, due to the lack of space, but a very simple application in which the regular parallelism is easy to be discovered.
available memory. Each process will send information about its CPU and memory availability in a collective communication step. Each host will examine the load of all other hosts and decide whether he can receive/send machines from/to other hosts.

The application has been evaluated in both a Myrinet and Ethernet networks using a JPEG image (1.26Mb, 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>
<thead>
<tr>
<th>TABLE 2</th>
<th>PRELIMINARY RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of hosts</td>
<td>No. of instances per host</td>
</tr>
<tr>
<td>Sequential*</td>
<td>1</td>
</tr>
<tr>
<td>Multi-threaded*</td>
<td>1</td>
</tr>
<tr>
<td>Ethernet</td>
<td>4</td>
</tr>
<tr>
<td>Myrinet</td>
<td>4</td>
</tr>
</tbody>
</table>
| * Not using the framework

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 DEVELOPMENT

This paper introduces a new distributed architecture based on state machines concept. The framework has been designed especially for application requiring data parallelism.

The architecture proved to be scalable, flexible and reduces the development time considerably by providing an engine for state machines management and migration. Load balancing support helps a lot in improving the performance and reliability of the system.

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 features which bind perfectly with MPI .Net.

Preliminary results were obtained in a high performance network (Myrinet network) as well as in a common Ethernet network. The results can be significantly improved by developing a solution to overcome the main bottleneck: data transfers. The data transfers influence significantly the time needed for processing.

The solution would be to integrate the framework with an independent distributed file system whose only goal is to handle data availability over the entire distributed environment by replicating data over multiple hosts and also provide data location information for the state machines framework as it is more convenient to migrate processes towards data than the other way around.

REFERENCES