The Evolutionary Design of a Framework for Computational Steering

Cosmin M. Poteraş*, Călin Constantinov*, Mihai L. Mocanu*

** Faculty of Automation, Computers and Electronics, University of Craiova, RO-200440 Romania*
(e-mail: { cpoteras, mmocanu}@software.ucv.ro)

Abstract: Computational steering aims both to interfere with an otherwise autonomous computational process, to change its outcome, and to enable the discovery of new features of the computational processes through integrated experiments. Traditionally, computational steering has been applied to large, compute-intensive and non-interactive simulations, where it more specifically refers to the practice of guiding a simulation experiment into some region of interest. In this paper, we review the motivations for computational steering and introduce an evolutionary design for a framework that takes into consideration two of its main important aspects, *program steering* and *data steering*, together with an approach for static scheduling based on a genetic algorithm. We then outline the capabilities of the framework by simulating the execution for three categories of applications, with low, medium and high communication needs, under two running scenarios – with and without tasks migration (required remote data will always be transferred). The results showed that program steering could bring more benefits in the given setting than data steering, and that there is a reasonable loss of efficiency between 16 and 64 processors, which could be explained correlated to the loss in data transfers gain.

Keywords: Computational Steering, Modeling, Simulation, Visualization, Genetic Algorithm.

1. INTRODUCTION

Modeling and simulation have become key phases for a wide spectrum of applications in modern research in all computer science areas. Cluster and grid simulation applications that employ parallel computing techniques (i.e. MPI, OPENMP) to simulate real processes are just a common example (Watanabe, 2011). Modeling, as a general term, denotes a 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. When the study involves experiments on the model (or, to be more specific, numerical evaluation of the model behavior, using the computer, under various hypotheses and working scenarios), we call that (numerical) computer simulation.

Traditionally, computer simulations are computationally intensive, non-interactive and slow – if they aim towards the obtaining of meaningful results with a high degree of confidence (result accuracy is conditioned by the granularity of the model, which should be adjusted up to a useful "scale", sometimes in iterative steps). The traditional steps in simulation are to construct a model, prepare input, execute a simulation, and then visualize the results. A text file describing the initial conditions and parameters for the course of a simulation is prepared, and then the simulation is submitted to a batch queue, to wait until there are enough resources available to run the simulation. The simulation runs entirely according to the prepared input file, and outputs the results to disk for the user to examine later. However, more insight and a higher productivity can be achieved if intermediary inspections of executions are allowed, and the model could be adjusted accordingly. This is the underlying idea of *computational steering*: researchers change parameters of their simulations on-the-fly and may receive feedback on the effect. In this way simulations may be executed in an interactive manner. This could be a simple matter, as allowing the user to monitor the values of some parameters in their simulation and, if necessary, to edit the values of others, or a more complex issue, that would require the integration of efficient infrastructures and good computational techniques. Anyway, a supplemental and useful benefit will be in that the researcher running a steerable simulation obtains an intuitive understanding of its behavior, i.e. the correlation between the modifications and the reactions of the process.

Computational steering can be viewed simply as a process of manual intervention on an autonomous computational system, with the goal to analyze and modify outputs, in order to increase its efficiency. But apart from its 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). 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 (Gu et al, 1994; Allan and Ashworth, 2001); however not many of the projects led to practical tools.

The rest of the paper is organized as follows. Section 2 describes background and related work in the domain, that is, similar systems and useful ideas. The next section reviews the initial design and a few implementation details for the State Machines Based Distributed (Sub)System (Poteras and Mocanu, 2011). Section 4 describes the evolutionary design aimed to accommodate the conceptual model for a Distributed Chunks Flow Management (Sub)System (Mocanu and Poteras, 2011). Section 5 explains how the two subsystems work together and presents integrative experimental results based on a *genetic algorithm* for static scheduling. The last section concludes the paper and presents some future work ideas.

2. BACKGROUND

Some of the most relevant frameworks for distributed simulation and computational steering, for the scope of this paper, may be considered: COVS, RealityGrid, CUMULVS and CSE. COVS, or Collaborative Onlline Visualization and Communication (Riedel et al, 2008) 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. The RealityGrid (Jha et al, 2004; Brooke et al, 2003) 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, or Collaborative User Migration User Library for Visualization and Steering, 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, or Computational Steering Environment (van Wijk, 1997; van Liere, 1997) 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 (Cohen, 2008) 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 are mainly systems that transfer files in distributed environments in raw format without any logical partitioning of the data, and they might be a good choice for distributed environments, especially for those based on slower networks. However, the main drawbacks 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.

Rather than relying on hardware to deliver the highest availability, the Hadoop Distributed File System was designed to detect and handle failures at the application layer, by this delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures. Hadoop is a software library conceived as part of the Apache Hadoop (http://hadoop.apache.org/) distributed systems framework. It has been built upon the Google's Map-Reduce architecture as well as HDFS file system, which 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 (Allan and Ashworth, 2001). The main drawback of HDFS seems to be, again, the centralized architecture built around the *name* node. Failure of the *name* node implies failure of the entire system.

Due to the 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 looked for support for load balancing and tasks migration, and safety features. For this we concluded to a form of design known as *evolutionary design*. Essentially, evolutionary design is a way to construct a system in which the design grows as the system is implemented. In other words, design is part of the programming processes and the design changes as the program evolves.

The overall objective of our design is to merge 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 several iterations of the actual design of the framework that consider both main and important aspects, program steering and data steering, together with an approach for static scheduling based on a genetic algorithm. Besides the overall testing of the resulting system, we'll prove the capabilities of the framework by simulating and optimizing the execution for several categories of applications, with low, medium and high communication needs, under different running scenarios – with and without tasks migration, but with appropriate transfer of the required remote data.

Thus, the term *evolutionary design* gets another connotation which is also within the scope of this paper. Evolutionary design, applied to algorithmic development, incorporates a computational model of evolution and natural selection that has been successfully applied to many complex optimization problems with nonlinear, temporal or stochastic components, where traditional optimization techniques proved to be inadequate.

3. FRAMEWORK INITIAL DESIGN AND IMPLEMENTATION

The enabling practices of continuous integration, testing, and refactoring, provide a new environment for the evolutionary design. In order to make this work, one should define clearly not only the goals, but also the restrictions of the future system. Many domains impose strict requirements for software in execution, calling for very high safety standards as well as high performance environments, being simply incompatible with errors and instability; there may dramatic effects associated, for instance, with an error in a software application that assists a surgery. To improve the reliability and safety, one has to make sure that in any moment the software is in a consistent state; this might need the analysis of all possible states prior to the system development, making sure the system's reaction is appropriate in any 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.

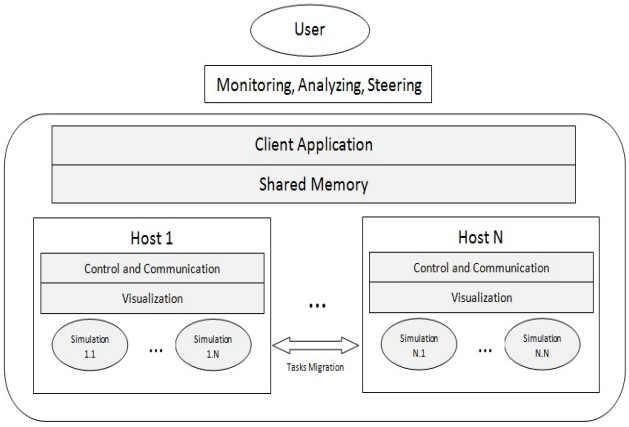


Fig. 1. The structure of the proposed distributed system.

Tasks migration together with live monitoring of the distributed environment reveals new possibilities for defining dynamic load balancing algorithms. We aimed initially to a new design model for a distributed simulation framework (environment) 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, which 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 function $\delta: \Sigma \times S \rightarrow S$ and F is the set of final states. The architecture 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. 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 state. 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) its 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.

- Define system parameters (implementing IParameter)
- Define 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.
- Instantiate the TransitionTable class and populate it with mappings of type <<parameter, current state>, future state>

The StateMachinesManager will be instantiated and run on each host. Fig. 2 illustrates the design and also the workflow within the system.

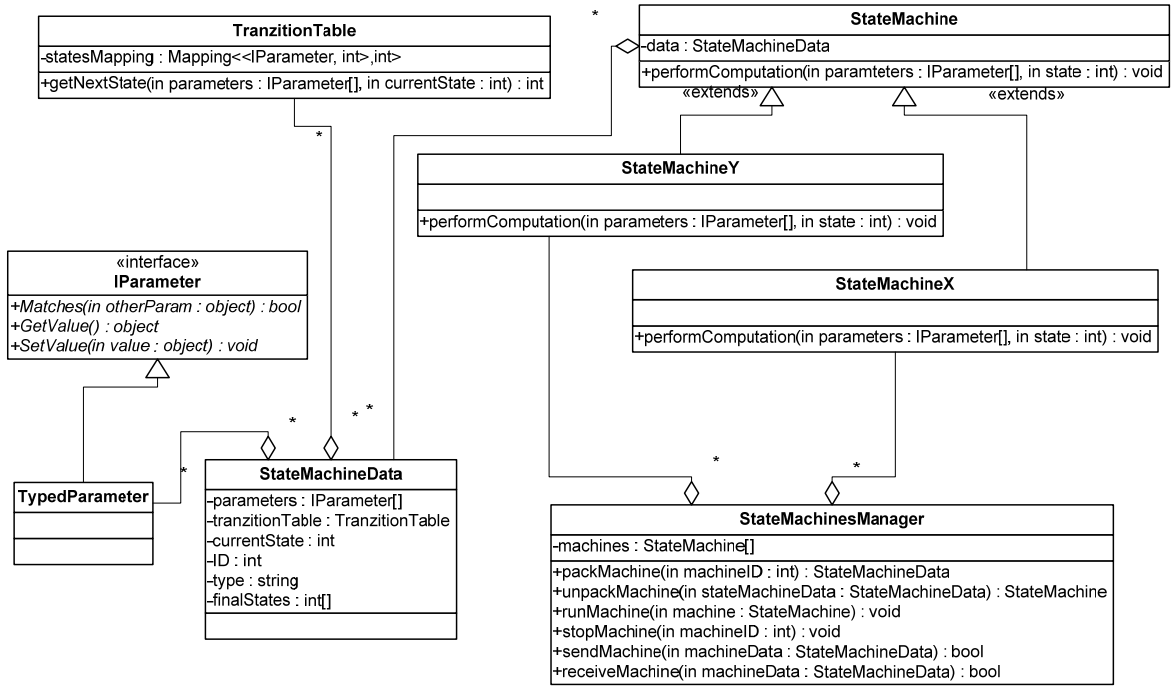


Fig. 2. The library's initial architecture and workflow

4. DESIGN EVOLUTION: DISTRIBUTED CHUNKS FLOW MANAGEMENT SYSTEM - THE CONCEPTUAL MODEL

As we usually consider it, a design is a mapping process from the design requirements to a design result. When the design requirements are modeled in a functional space, design results may be found either in an attribute space or, more closely to the evolutionary approach, in a parameter space that evolves together with the model. The completion of the model for data to describe design requirements and the design results developed at different design stages, from conceptual design to detailed design, is corresponding to the design descriptions at different design stages.

For the proposed data flow management system, whose model is illustrated in Figure 3, we started from a series of "autonomous" features and then we looked for the integration requirements, with the framework described in the previous section. The main features depicted in the initial stages of design were:

- *cost:* better price/ performance ratio can be obtained as long as commodity hardware is used for the component computers;
- *predictability*: the system must provide the desired responsiveness in a timely manner;
- *portability:* cross-platform system design that does not require special system privileges for running):
- *extensibility:* new data partitioning modules can be integrated at runtime);
- *scalability:* hosts can be added at run-time; storage capacity, the size of the network or the overall load on the system can be increased, and this should not have a significant effect;
- *run-time data consistency* (synchronization): this is the ability of the system to coordinate actions of multiple components (this underlies the ability of a distributed system to act like a non-distributed one);
- *abstract communication API*;
- *customizable data handling* for all data types, etc.

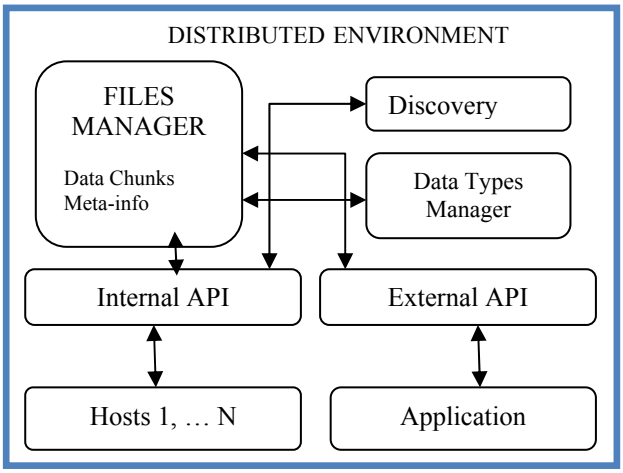


Fig. 3. The data flow management system design model

The data flow management system is intended to separate data flow management from the processing level the framework provides, while maintaining *high data availability* and, possibly, *fault tolerance*, which is in our understanding the capacity to recover from component failures without performing incorrect actions. The entire model has been built around one key element, the *data chunk*. It usually represents a file partition but it can also be any data object required by the application's processes. Besides the data piece itself, a *data chunk* also contains meta-information describing the data piece, like: size, location inside source file, the data type, timestamp of latest update or the class that handles chunks of its type. Thus, the most important contribution of the data flow management system is the way it handles chunks of different types in an abstract mode without actually knowing what is inside the chunk, leaving the data partitioning up to the application level. This is very important from an application perspective, allowing it to map data chunks to processing tasks very efficiently. No restrictions are imposed by the data flow system on data partitioning. The bridge between the abstract representation of data chunks and their actual type is the Type Manager. It is able to make use of external classes (defined at the application level) where all the file type specific functionality can reside. The classes are dynamically loaded whenever the application layer needs partitioning, files reconstruction as well as information related to the collection of chunks (i.e. the number of chunks). It is the applications' developer task to implement the data chunks handler classes. The data flow management system only provides a set of interfaces that help to implement the partitioning logic. For example, one might need to handle two types of files in their distributed application: image files and text files. In case of the image files a data chunk might be represented by a rectangular region of the initial image. Multiple such chunks can cover the entire image. An image can be split into rectangular chunks by dynamically invoking the image partitioning method. In case a node needs an entire file that is spread all across the system, the data flow system can acquire all its chunks from different hosts and recompose the image by dynamically invoking the image reconstruction method. In case of a text file, the chunks can take the form of paragraphs, or pages, or simply and array of characters of a certain size. In a similar way the files can be dynamically partitioned and reconstructed. Later in this paper we will discuss the development effort involved in writing such classes.

Data scalability and synchronization

As we mentioned before, the data flow system must be able to scale up dynamically at run time without using a central node. This functionality is achieved by the Discovery Unit which broadcasts and listens to discovery messages. There are two API interfaces that allow the data flow management system nodes to communicate with each other and also with the client application.

A key feature in any distributed system that handles large amount of data is keeping data synchronized. Spreading

data around the network while keeping it up to date uses events. Each node that has updated a data chunk must broadcast to all other nodes that he is aware of about the changes, and event handlers update the timestamp of the affected data. Depending on the nodes connectivity there are two choices:

1. No event retransmission – the ideal situation when the network bandwidth allows 1 to 1 connections between any two nodes in the system; it is enough to broadcast an update event once to all other nodes in the system.

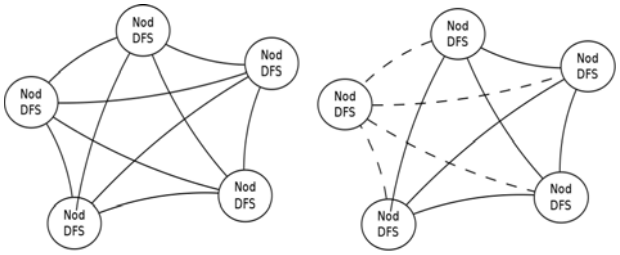


Fig. 4. Retransmission not needed for event propagation

2. Event retransmission – when there is at least one node not interconnected with all other nodes in the system. To make sure that node is always notified about update events, retransmission is necessary; to stop infinite loop of update events nodes employ timestamps (whenever an update event time stamped in the past it will be ignored)

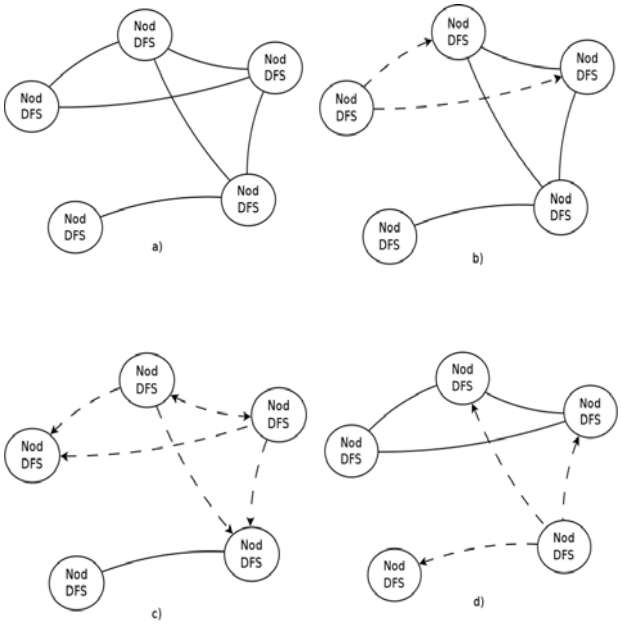


Fig. 5. Events retransmission

A data flow scenario

The data flow algorithm is based on availability tables. Let us analyze briefly a concrete scenario. If we assume that the data flow management system consists of nodes N_0, N_1 … N_n, let node N_0 be interested in aquiring data chunks C_1, C_2, … C_m. N_0 will broadcast a request for C_1,...C_m to the entire system.

```
N0 THREAD 1
procedure HandleChunkRequest(P0, ..., Pm)
    BroadcastChunkRequest(P0 ,... , Pm)
    while  exists Pi not transferred, i=1,m
      if available[Pi , Hostj] = true, j=1,m then
            AcquireThreaded(Pi,Hostj)
        wait(response)

N0 THREAD 2
procedure ReceiveChunksInfo(found, Host)
    if found[Pi] = true,  i=1,m then
        available[Pi,Host] = true
        notify(raspunsNou)
```

Algorithm 1. Data flow

Nodes N_1...N_n reply back to N_0 with a subset of C_1,....C_m that they host As soon as the replies arrive, N_0 builds a chunks availability matrix having as rows the nodes N_1....N_n and as columns chunks C_1...C_m. (N_i,C_j) gets valued 1 if the chunk C_j is available on host N_i, otherwise it gets valued 0. N_0's main goal is to establish as many connections as possible, but not more than one connection per serving host (at most n-1 connections at a time). Chunks availability responses are performed in an asynchronous manner so that N_0 won't have to wait for all responses before proceeding with transfers. Instead it will establish connections as the responses arrive, overlapping chunks transfer with availability requests. Whenever a chunk transfer completes, the External API will be informed about it and the client application can start processing the newly acquired data. As chunks might spread across the system while N_0 transfers its chunks, the availability matrix will be constantly updated by sending new availability requests whenever a chunk transfer completes and N_0 has established less than n-1 connections (free download slots available).

The Algorithm 1 describes the data flow.

Data partitioning and support for load balancing

As previously mentioned, the user is able to retrieve exactly the data of interest causing an important reduction of the amount of data that travels through the network. A data chunk is basically any logic unit of data extracted from a data set (usually a file) according to a certain algorithm that reflects the application's needs. The data extraction is based on the most simple principle: request – answer. The application places queries against the data flow management system, queries are broadcasted in the entire system, each node invokes the appropriate chunker (the one associated with the request's type), the chunker extracts the logical piece of data according to its internal algorithm (custom algorithm designed to serve the application environment's needs), and ultimately it replies back with the data chunk.

In distributed applications it often happens that the processing of a data chunk requires less time than the transfer of the data itself. For this reason it might be a good practice to migrate the processing task towards the

data than transferring data to the processing host. DATA FLOW MANAGEMENT SYSTEM is able to provide through its external API locating information about the data it holds (data aware system). It is the application's task to query the system for data location information and migrate the processing tasks throughout the nodes in order to reduce or eliminate the data transfer time.

Developer's task: Implementing Chunker classes

Chunker classes define how files or data objects are split into data chunks. A chunker class is a class that implements a Chunker interface defining the following content: requests structures, Response structures, Data Requests handlers, Meta-Data Requests handlers (ensuring data-awareness). An important feature of the system is that not all nodes need to hold all chunker classes known by the system. They only need the chunkers associated with the data they serve. If unknown type of data is requeste the node can dynamically load its associated chunker ar run-time.

5. EXPERIMENTS AND RESULTS

Figure 6 illustrates the final design, resulted from the integration of the two subsystems previously described: the state machine based distributed system (SMBDS) and data chunk flow management system (DCFMS).

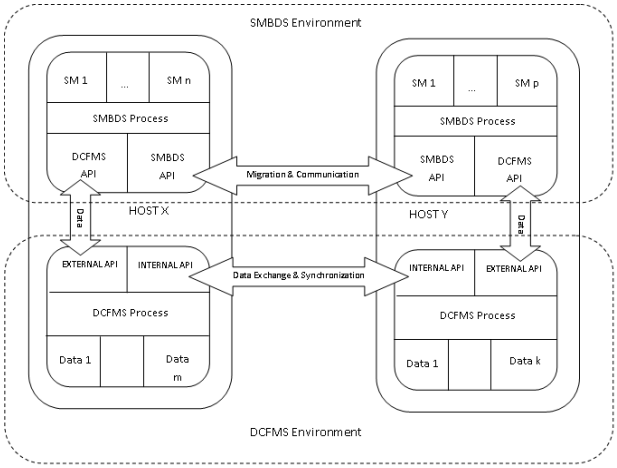
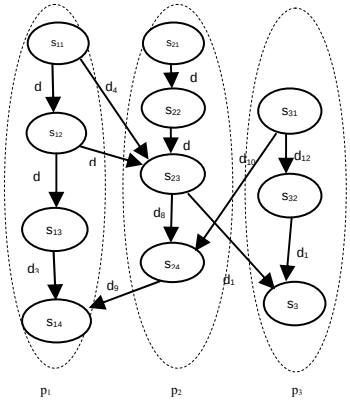


Fig. 6. The result of the evolutionary design: SMBDS-DCFMS integration

Ideally each host would run one instance of each system. However this is not a constraint. For maximum performance it is recommended that each SMBDS process should be bound to a local DCFMS process. The former subsystem will manage the execution of state machines, acquiring data for their execution via the external API interface of the second subsystem. Its processes interact with the DCFMS process (through an external API) in order to acquire input data for each state of the machines. At the same time SMBDS procesess will interact with each other for migrating tasks according to localization information offered by DCFMS, whose processes also interact between them as discussed in section 3. Once integrated, a new issue appears, namely scheduling the execution of the state machines as well as the transfers between them so that the causality constraints between machines states is accomplished and the execution time is as short as possible. The states of the state machines can define input-output dependencies between them as no state execution can be performed until all input data has been acquired. As a consequence there can be considered two types of causality constraints: internal (between the states of each machine) and external (between the states of different machines).

For a better understanding of the scheduling algorithm we will consider a state machine as being a task composed by many subtasks and we will try to schedule the subtasks so that they fulfill the causality constraints. Scheduling subtasks instead of tasks is perfectly possible due to the migration of state machines. By using migration of state machines towards data, one can reduce the number of transfers through the network. In (Wang et al, 1997) there has been defined a genetic algorithm for scheduling subtasks in distributed environments. The model used is slightly different than the one introduced in this paper. The main differences are related to the number of processors per machine which is at most one, the network topology which uses a crossbar switch, and the transfer strategy which uses serial transfers for each network link. Instead, DCFMS uses the Ethernet model, allowing the network link to be used in multiple transfers at the same time from multiple sources while SMBDS aims to be suitable for multiprocessing machines. These differences are reflected mainly in the evaluation phase of the genetic algorithm. For this reason we will make use of the selection, crossover and mutation phases as they are presented in (Wang et al, 1997), while the evaluation has been redesigned so that it fits the integrated model.

If we let S be the set of subtasks of all state machines and |S| the number of such subtasks, then $S = \{s_i, 0<=i<|S|\}$. Let P be the set of available processors $P= \{p_j, 0<=j<|P|\}$, where |P| is the number of processors. The set of data objects defining dependencies between subtasks would be D, where $D=\{ d_k, 0<=k<|D| \}$ and |D| is the number of data objects. We also introduce the available machines which might own a subset of P as being $M = \{ M_i \subseteq P \mid \forall i<>j, M_i \cap M_j = \phi, 0<=i,j<|M|\}$, where |M| is the number of available machines.



$S = \{s_{11}, s_{12}, s_{13}, s_{14}, s_{21}, s_{22}, s_{23}, s_{24}, s_{31}, s_{32}, s_{33}\}$
$|S| = 11$

$D=\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}, d_{12}, d_{13}\}$
$|D| = 13$

$P = \{p_1, p_2, p_3\}$
$|P| = 3$
$M = \{M_1 M_2,\}, M_1=\{p_1\}, M_2=\{p_2,p_3\}$

Fig. 7. The execution model

In Fig. 7 we illustrate an execution example for three state machines scheduled for execution on three different processors. We may notice here the internal and external causality constraints defined by the data objects. Data objects are modeled in the context of the data flow subsystem by logical partitioning of data. The next step is to define a chromosome structure. The chromosome in our case represents a complete schedule for a set of subtasks. The chromosome will be composed by a matching string and a scheduling string. The matching string defines a mapping between the available processors while the scheduling string defines the order of execution for each state. An example of a chromosome is presented in Fig. 1. There are available 5 processors that need to execute 7 subtasks.

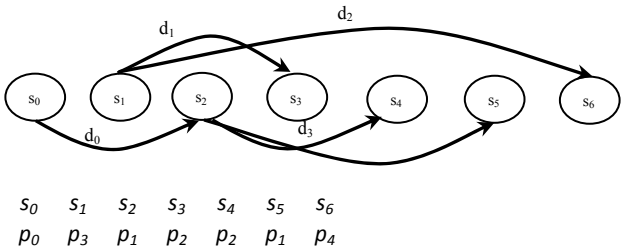


Fig. 8. Chromosome example

For evaluating the fitness value of a chromosome one needs to evaluate the execution time of each subtask as well as the size of each data object, before the scheduling. This is a common practice in the scheduling research field (Freund, 1994; Singh and Youssef, 1996).

Before proceeding with the chromosome evaluation we will make the following assumptions: each machine can perform multiple transfers at the same time on both input and output lines, the transfer time for a locally available data object is considered to be zero, all machines use network links with the same bandwidth (considered to be of two units) for both input and output, the bandwidth is shared equally among all running transfers.

In the remainder of this section we will analyze an example of the chromosome in Fig. 8. Table 1 defines the size of each data object while Table 2 defines the execution time for each of the subtasks on every processor.

Data	Size
d_0	5
d_1	2
d_2	6
d_3	3

Table 1. Data size

M	P	s_0	s_1	s_2	s_3	s_4	s_5	s_6
M_0	p_0	3	4	2	5	3	2	1
M_1	p_1	4	2	5	4	3	2	3
	p_2	2	4	4	3	3	2	4
M_2	p_3	3	2	3	2	2	4	3
M_3	p_4	4	5	6	8	7	5	6

Table 2. Execution time

As mentioned before each subtask will be scheduled as soon as its data objects become available. A data object will be transferred towards its associated subtasks as soon as it becomes available. The scheduling of subtasks and data transfers for the chromosome in Fig. 8 is presented in Fig. 9.

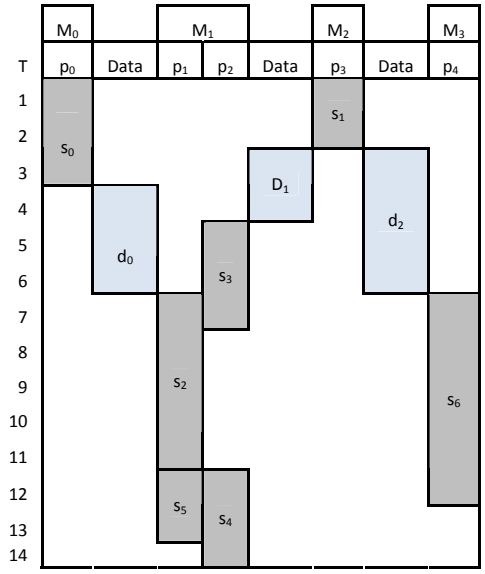


Fig. 9. Chromosome scheduling

Subtask s_1 ends at moment T_2 and produces d_1 of size 2 needed for s_3 to start its execution and d_2 of size 6 needed for s_6's execution. For the next two time intervals d_1 and d_2 will share the output bandwidth of M_2 equally (transfer speed of 1 unit). At moment T_4, d_1 will finish transferring as the simultaneous transfer of d_0 from M_0 does not affect the download speed of M_1 as it is already limited to one unit due to the shared output bandwidth of M_2. Even if M_0 can upload data with full bandwidth (2 units), M_1 can increase the download speed only after T_4. The same applies for object d_6. The fitness value of the chromosome in figure 8 will be 14.

Using the above presented algorithm, the state machine based initial framework design, completed with the data flow management system, has been evaluated. There will be considered two scenarios: running applications by taking profit from tasks migration and running applications without any migration. In the first scenario there will be used all features of both (sub) systems. The second scenario requires the tasks to be equally distributed across the environment (in arbitrarily order) without any kind of migration (required remote data will always be transferred).

The results of the two scenarios are to be compared. There have been considered three categories of applications depending on their needs for communication:

- *low communication applications* ($0 \le |D| \le (1/3)\ |S|$)
- *medium communication applications* ($(1/3)\ |S| \le |D| < (2/3)\ |S|$)
- *high communication applications* ($(2/3)|S| \le |D| < |S|$)

The execution time for subtasks (each state of state machines) has been randomly picked up from the interval [1-10]. The size of data objects has been picked up from the interval [5-20]. The network speed on both input and output has been considered to be equal to 1.

We considered two computational environments: the former is composed of 8 machines with 2 processors each, and the latter is composed of 32 machines with 2 processors each. The algorithm was executed using the "elite chromosome" strategy which keeps an elite chromosome that is updated as better chromosomes are discovered. The algorithm stops after 150 cycles without improving the elite chromosome. For each testing scenario have been run 100 execution configurations. The final result has been considered to be the average of all results. There have been considered a number of 300 subtasks (states). It is known that each data object defines at least one dependency between two subtasks. We will call *number of additional dependencies* the number of dependencies defined by a data object excluding the first (basic) dependency. The total number of additional dependencies (across all data objects) will be picked randomly from the interval [1%-35%]. The distribution of additional dependencies across data objects will also be generated randomly. Table 3 presents the execution time for each scenario, while Table 4 presents the amount of data transferred in each scenario.

No. of states	16 proc. / 8 machines			64 proc / 32 machines		
	Exec time - migration	Exec. time no migration	Gain %	Exec. time migration	Exec. time no migration	Gain %
1 – 100	145.00	219.90	34.06	89.55	130.90	31.58
101-200	246.45	351.10	29.80	146.55	199.35	26.48
201-300	369.15	482.45	23.48	202.10	255.40	20.86

Table 3. Execution time results.

No. of states	16 proc. / 8 machines			64 proc. / 32 machines		
	Amount of data - migration	Amount of data – no migration	Gain %	Amount of data – migration	Amount of data – no migration	Gain %
1 – 100	782.20	898.05	12.90	871.04	943.20	7.65
101-200	1762.46	1963.07	10.22	2342.24	2496.25	6.17
201-300	3258.20	3580.73	9.00	3817.17	4010.90	4.83

Table 4 –Transferred data results

Applications with low communication have employed between 1 and 100 data objects. We noticed a loss of computational gain between the 8 machines system and the 32 machines system that can be explained correlated to the loss in data transfers gain; allocating subtasks on many machines requires data to be spread across a wider environment which causes more transfers on the network and implicitly causes delays on the computational level. The same applies to the other two categories of applications. Another conclusion based on the two tables is that the computational gain and transfers gain reduce as the number of subtasks increases. This was somehow predictable; the more data objects involved in the execution, the more transfers are to be scheduled and the more delays are to be caused at the computational level.

6. CONCLUSIONS AND FUTURE WORK

In this paper we reviewed the motivations for computational steering and introduced an evolutionary design which considers two important aspects, *program steering* and *data steering*, and tries to integrate them into a unique framework. The initial design was for a distributed (sub) system for managing computational tasks represented as state machines. It proved enough robust, scalable and flexible, able to accommodate the conceptual model for a second (sub) system, for data flow management across the distributed environment; this works with *chunks* of different types in an abstract mode, without actually knowing what is inside the chunk, thus leaving the data partitioning up to the application level. This is very important from an application perspective, allowing it to map data chunks to processing tasks very efficiently. The integration of the two systems has been based on API interfaces. The experimental section of the paper illustrates how the two subsystems work together, how it can reduce in a considerable manner the development time and presents integrative results based on a genetic algorithm for static scheduling.

State machines scheduling has been performed at states level by using a genetic algorithm. Experimental results showed considerable computational gain especially in low communication applications where the execution time can be improved by up to 34% while the amount of data transferred reduced with up to 12.9% related to non migrating executions of the same workload.

Our future work intentions will consider the development and inclusion in the combined framework (SMBDS – DCFMS) of several algorithms for load balancing and dynamic allocation of computational resources. Making these algorithms "reactive" to available computational resources, on one hand, and to small changes in data distribution across the virtual computational environment, on the other hand, could lead to improved predictable performance, compared to the performance attained in the case of static allocation and presented in this paper. We intend to address in a more detailed survey paper the development of dynamic load balancing of computational resources combined to dynamic data replication techniques.

Up to now, the effort of innovation in the distributed data flow was focused to finding of new techniques of logical partitioning and to offering, as much as possible, complete information for data localization. Future work should be directed towards the optimization of transfer techniques among the nodes of a distributed system. Monitoring network traffic and the development of algorithms for network routes ranking could help deciding which route is the most appropriate route for faster transfers. The selection of an optimal dimension for data chunks that are transmitted across a network, together with caching techniques, might improve considerably the availability of data at the level of computational nodes. Employing probabilistic algorithms for simple (but not simplistic) replication of data and state machines may

bring more efficiency and stability to a system that, in our opinion, has a great potential of overall improvement.

The experiment and analysis fulfilled also convinced us on the potential benefits that a visual environment for programming and execution may bring. The design of a friendly GUI (graphical user interface) appropriate for the system described in this paper should allow: visual definition of state machines and interactions among them, easy intuitive definition and change of specific execution scenarios, visual monitoring of executions and, last but not least, joining and combining different techniques for computational steering.

REFERENCES

Allan R.J., Ashworth, M. (2001) *A Survey of Distributed Computing, Computational Grid, Meta-computing and Network Information Tools*, available from http://www.ukhec.ac.uk/publications/reports/survey.pdf

Brooke, J. M., Coveney, P.V. et al. (2003) Computational Steering in RealityGrid. In *Proceedings of the UK e-Science All Hands Meeting*

Cohen, B. (2008). The BitTorrent Protocol Specification, http://www.bittorrent.org/beps/bep_0003.html

Freund, R. F. (1994) The challenges of heterogeneous computing. In *Parallel Systems Fair* at the *8th International Parallel Processing Symposium.* IEEE Computer Society, 84–91, Cancun, Mexico.

Geist, G.A., Kohl, J.A., Papadopoulos, P. M. (1997) CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. In *Intl. Journal of High Performance Computing Applications*, 11(3), 224-236

Gu, W, Vetter J, Schwann, K. (1994) An Annotated Bibliography of Interactive Program Steering, *SIGPLAN Notices* 29, 140-148 (and Technical Report GIT-CC-94-15, Georgia Institute of Technology)

http://hadoop.apache.org (accessed oct. 2011) The Apache™ Hadoop™ project

Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D. (2007) Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks, *EuroSys - European Conference on Computer Systems*, Lisbon, Portugal, 59-72

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

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

Mocanu, M., Poteras, C., Petrisor, C. (2011) Improving Parallel Data Flow Support in a Visualization and Steering Environment, In *Recent Researches in Applied Informatics – Proc. 2nd International Conference on Applied Informatics and Computing Theory (AICT '11)*, 226-231

Poteras, C., Mocanu M. (2011) A State Machine-Based Parallel Paradigm Applied in the Design of a Visualization and Steering Framework, In *Recent Researches in Applied Informatics – Proc. 2nd International Conference on Applied Informatics and Computing Theory (AICT '11)*, 232-236

Riedel, M., Frings, W. et al. (2008) Extending the collaborative online visualization and steering framework for computational Grids with attribute-based authorization. In *GRID 2008*, 104-111

Singh, H., and Youssef, A. (1996) Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *Proceedings HCW'96 - Heterogeneous Computing Workshop*, 86–97, IEEE Computer Society, Honolulu, HI.

van Liere, R., Mulder, J.D., van Wijk, J.J. (1997) Computational steering. In *Future Generation Computer Systems*, 12(5), 441–450

van Wijk, J.J., van Liere, R. (1997) An environment for computational steering. In *G.M. Nielson, H. M¨uller, and H. Hagen, editors, Scientific Visualization: Overviews, Methodologies, and Techniques*, 89–110. Computer Society Press

Wang, L., Siegel, H., Roychowdhury, V., Maciejewski, A. (1997) Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach, In *Journal of Parallel and Distributed Computing* 47, 8–22

Watanabe, T. (2011) Numerical Simulation of Flow Field In and Around a Droplet in an Acoustic Standing Wave. In *Recent Advances in Fluid Mechanics and Heat & Mass Transfer*, 170-175, Florence, Italy