Toward a Computational Steering Environment based on CORBA

Olivier Coulaud, Michaël Dussere, Aurélien Esnard

Toward a Distributed Computational Steering Environment based on CORBA

O. Coulaud[a], M. Dussere[a] and A. Esnard[a]

[a]Projet ScAlApplix, INRIA Futurs et LaBRI UMR CNRS 5800,
351, cours de la Libération, F-33405 Talence, France

This paper presents the first step toward a computational steering environment based on CORBA. This environment, called *EPSN* [1], allows the control, the data exploration and the data modification for numerical simulations involving an iterative process. In order to be as generic as possible, we introduce an abstract model of steerable simulations. This abstraction allows us to build steering clients independently of a given simulation. This model is described with an XML syntax and is used in the simulation by some source code annotations. *EPSN* takes advantage of the CORBA technology to design a communication infrastructure with portability, interoperability and network transparency. In addition, the in-progress parallel CORBA objects will give us a very attractive framework for extending the steering to parallel and distributed simulations.

1. Introduction

Thanks to the constant evolution of computational capacity, numerical simulations are becoming more and more complex; it is not uncommon to couple different models in different ditributed codes running on heterogeneous networks of parallel computers (e.g. multi-physics simulations). For years, the scientific computing community has expressed the need of new computational steering tools to better grasp the complexity of the underlying models. The computational steering is an effort to make the typical simulation work-flow (modeling, computing, analyzing) more efficient, by providing on-line visualization and interactive steering over the on-going computational processes. The on-line visualization appears very useful to monitor and detect possible errors in long-running applications, and the interactive steering allows the researcher to alter simulation parameters on-the-fly and immediately receive feedback on their effects. Thus, the scientist gains a better insight in the simulation regarding to the cause-and-effect relationship.

A computational steering environment is defined in [1] as a communication infrastructure, coupling a simulation with a remote user interface, called steering system. This interface usually provides on-line visualization and user interaction. Over the last decade, many steering environments have been developed; they distinguish themselves by some critical features such as the simulation integration process, the communication infrastructure and the steering system design. A first solution for the integration is the problem solving environment (PSE) approach, like in SCIRun [2]. This approach allows the scientist to construct a steering application according to a visual programming model. As an opposite, CAVEStudy [3] only interacts with the application through its standard input/output. Nevertheless, the majority of the steering environments, such as the well-known CUMULVS [4], are based on the instrumentation of the application source-code. We have chosen this approach as it allows fine grain steering functionalities and achieves good runtime performances. Regarding the communication infrastructure, there are many underlying issues especially when considering parallel and distributed simulations: heterogeneous data transfer, network communication protocols and data redistributions. In VIPER [5], RPCs and the XDR protocol are used to implement the communication infrastructure. Magellan & Falcon [6] communicates over heterogeneous environments through an event system built upon *DataExchange*. CUMULVS uses a high-level communication infrastructure based on PVM and allows to collect data from parallel simulations with HPF-like data redistributions. In *EPSN* project, we intend to explore the capabilities of the CORBA technology and the currently

[1] *EPSN* project is supported by the French ACI-GRID initiative (grant number PPL02-03).

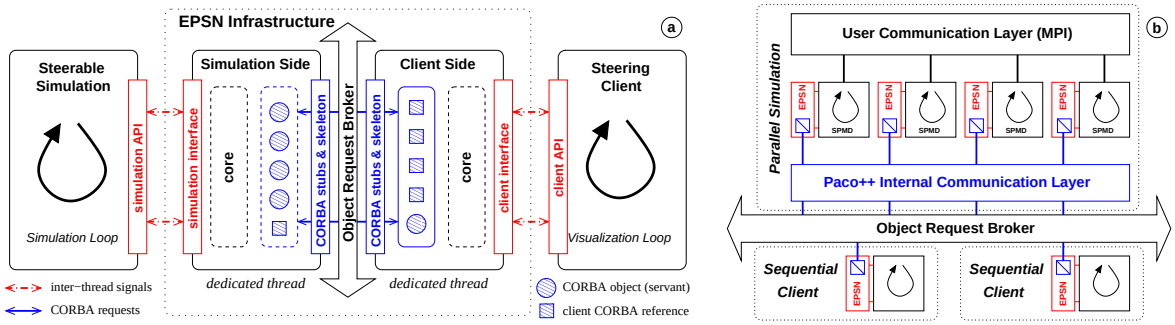

Figure 1. (a) Detail of the *EPSN* architecture. (b) *EPSN* parallel architecture with PaCO++.

under development parallel CORBA objects for the computational steering of parallel and distributed simulations.

In this paper, we first describe the basis and the architecture of *EPSN*. Then, we illustrate the integration process of a simulation. Finally, we present preliminary results on the *EPSN* prototype, called *Epsilon*.

2. The *EPSN* Environment

2.1. Principles

EPSN is a steering platform for the instrumentation of numerical simulations. In order to be as generic as possible, we introduce an abstract model of steerable simulations. This model intends to clearly identify where, when and how a remote client can safely access data and control the simulation. We consider numerical simulations as iterative processes involving a set of data and a hierarchy of computation loops modifying these data. Each loop is associated with a single counter and the association of all these counters enables *EPSN* to precisely follow the global time-step evolution. The "play/stop" control operations imply the definition of breakpoints at some stable states of the simulation. In practice, we can easily steer most of the simulations by simply instrumenting their main loop. The basic access operations consist in extracting and modifying the data. As these data alternate between coherent and incoherent states during the computation, it implies the definition of restricted areas where data are not accessible. Several data can be logically associated to define *a group* that enables the end-user to efficiently manipulate correlated data together. Moreover, we have extended the coherency definition for groups to guarantee that all group members are accessed at the same iteration. This model, which fits well parallel applications (SPMD), needs to define a global time extension to maintain the data coherency over coupled or distributed simulations (MPMD). Such a mechanism implies an explicit association of the loops and breakpoints of the different simulation components.

The representation in the abstract model is obtained by pointing out the relevant informations on the simulation. First the user describes the simulation elements in an XML file, then he connects the simulation with its representation. To do so, he annotates the source code with *EPSN* API in order to locate the elements that he has identified and to mark their evolution through the simulation process. The XML description also intends to lighten and clarify this annotation phase.

2.2. Architecture and Communication Infrastructure

EPSN is a distributed and dynamic infrastructure. It is based on a client/server relationship between the steering system (the client) and the simulation (the server) which both use *EPSN* libraries. The clients are not tight coupled with the simulation. Actually, they can interact on-the-fly with the simulation through asynchronous and concurrent requests. According to this model, different steering systems can concurrently access the same simulation and, reciprocally, a steering system can simultaneously access different simulations. These characteristics make *EPSN* environment very

flexible and dynamic.

The communication infrastructure of *EPSN* is based on CORBA, but it is completely hidden to the end-user. CORBA enables applications to communicate in a distributed heterogeneous environment with network transparency according to a RPC programming model. It also provides to *EPSN* the interoperability between applications running on different machine architectures. Although, CORBA is often criticized for its performance, some implementations are very efficient [7].

The principle of *EPSN* infrastructure is to run a permanent thread attached to each simulation process. This thread contains a multi-threaded CORBA server dedicated to the communications between the steering clients and the simulation. As shown on figure 1(a), the simulation thread consists of different CORBA objects corresponding to *EPSN* functionalities. For being fully asynchronous, *EPSN* uses *oneway* CORBA calls and the client thread also implements a *callback* object to receive data from the simulation. In other steering environments, like CUMULVS, the simulation is in charge of the communications that occur during a single blocking subroutine call. In *EPSN*, the thread accesses directly to a data through the shared memory of the process without any copy. Between a process and the *EPSN* thread, the communications use standard inter-thread synchronization mechanisms based on semaphores and signals. This strategy combined with the asynchronous CORBA calls allows to overlap the communications.

In order to maintain a single representation of the simulation, we use a specific CORBA object, the *proxy*, running on the first simulation process. This object provides the description of the whole simulation and all the CORBA references needed by both the client and other simulation processes. As the proxy is registered to the CORBA naming service, remote clients can easily connect it. In order to achieve coherent steering operations on SPMD simulations, we have developed some protocols to synchronize the simulation processes. This synchronization implies to broadcast the request to all the involved processes and to synchronize on the first breakpoint before achieving the parallel request. To reduce the synchronization cost, the parallel processes can also synchronize once at the beginning and keep going synchronously after that. The parallel extension of CORBA objects, like PaCO++ [8], reduces the synchronization cost thanks to a better use of the parallel infrastructure. PaCO++ typically exploits an internal communication layer based on MPI (Fig. 1(b)). It also greatly eases the *EPSN* extension to parallel clients and for the inherent problem of data redistribution. On-going works are focusing on the integration of PaCO++ in *EPSN* and especially on the full support of regular data decomposition.

2.3. Functionalities

EPSN consists in two C/Fortran libraries, the first one provides functions to build a steerable simulation and the second one enables to build a remote steering system.

Control. The insertion of `barrier` function calls, acting as debugger breakpoints, in the simulation source code allows the user to control the execution flow of the simulation. The breakpoints can be remotely set "up" or "down" (`setbarrier` command) and they allow classical control commands (`play, step, stop`). Moreover, the calls to `iterate` function point out the evolution of simulation through the loop hierarchy.

Data Extraction. On the client side, the user can remotely extract data from the simulation by calling `get` functions. The client manages such a request with `wait` and `test` MPI-like functions. On the simulation side, data access is protected by `lock/unlock` functions, which delimit the "coherence areas" in the source code. Therefore, data sending can be done immediatly when receiving a get-request or delayed if the data is not accessible yet. Once the data is received by the client, it can be automatically copied in the client memory within the same `lock/unlock` areas as for the simulation, or it can start a treatment defined by the user thanks to a *callback function* call.

The user can also request a data *permanently*, with the `getp/cancelp` functions, in order to continually receive new data releases and produce "on-line" visualization. In this case, an acknowledgment system automatically regulates the data transfer from the simulation to the client, by voluntary ignoring some data releases to avoid to congest the client. Nevertheless, a `flush` function can be used to force the data sending at each timestep. Thus, it prevents the client to miss any release, but it can slow down the simulation according to the client load.

Data Modification. In the same way, the client can modify a data by calling the `put` function

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE simulation SYSTEM "Epsilon.dtd">

<simulation name="spray" context="parallel">
<nameservice host="localhost" port="2809"/>
<control running="true" >
  <breakpoint id="begin" state="down" location="specific"/>
  <loop id="loop" state="up"/>
    <breakpoint id="mid" location="distributed"/>
  </loop>
</control>
<data>
  <group id="mesh">
    <scalar id="nbNodes" type="long" access="readonly" location="replicated"/>
    <scalar id="nbTri" type="long" access="readonly" location="replicated"/>
    <sequence id="NodesCoord" type="double" location="replicated">
      <dimension size="nbNodes"/>
      <dimension size="2" />
    </sequence>
    <sequence id="Cells" type="long" location="replicated">
      <dimension size="nbTri"/>
      <dimension size="3" alloc="4"/>
    </sequence>
    <sequence id="Energy" type="double" location="distributed">
      <dimension size="nbNodes" decomposition="block"/>
    </sequence>
  </group>
</data>
</simulation>
```

Figure 2. *FluidBox* short XML description.

```
! --- Simulation Initialization ---
CALL ReadMesh(Mesh,MeshFile)
CALL Init(Data,Mesh,Var)
! --- Epsilon Initialization ---
CALL epsilon_init('simu.xml',numproc,nbprocs,ierr)
CALL epsilon_publish('nbNodes',Mesh%Npoint,ierr )
CALL epsilon_publish('nbTri',Mesh%Nelemt,ierr)
CALL epsilon_publish('NodesCoord',Mesh%coor(1,1),ierr)
CALL epsilon_publish('Cells',Mesh%nu(1,1),ierr)
CALL epsilon_publish('Energy',Var%Ua(1,1),ierr)
CALL epsilon_publishgroup('mesh',ierr)

IF (numproc.EQ.0) THEN CALL epsilon_barrier('begin',ierr)
CALL MPI_Barrier(MPI_COMM_WORLD,ierr) ! Barrier on master process
CALL epsilon_unlockall(ierr);
DO kt = kt0+1, ktmax ! Simulation loop
   CALL Inject(Data,Mesh,Var) ! Fluid injection
   CALL epsilon_barrier('mid',ierr) ! Barrier distributed on all processes
   ! --- Modify field values inside locked area ---
   CALL epsilon_lock('Energy',ierr)
   Var%Ua(:,:) = Var%Un(:,:)
   CALL epsilon_unlock('Energy',ierr)
   CALL Post(Mesh,Var)
   CALL epsilon_iterate('loop',ierr)
END DO

! --- Simulation ending ---
CALL WriteResult(Data,Mesh,Var)
CALL epsilon_exit(ierr)
```

Figure 3. *FluidBox* instrumented pseudo-code.

which transfers datas from the client memory to the simulation.

3. Building a Steerable Application

In this section, we present the integration of *EPSN* steering functionalities in a parallel fluid flow simulation software, *FluidBox* [9]. This MPI Fortran code is based on a finite volume approximation of the *Euler* equations on unstructured meshes and simulates a two-fluid spray injection. We detail the XML description of this simulation in the abstract model, the instrumentation of the source code and the different solutions proposed in *EPSN* to construct steering systems.

3.1. XML Description of the Simulation

The first phase in the construction of a steerable simulation consists in its description through an XML file. This description is the representation of the simulation shared by the simulation thread and by the client thread. On the simulation side, the XML is parsed at the initialization of *EPSN* to build all the necessary structures and to parameter the instrumentation. The clients dynamically get this description from the simulation so they do not need a direct access to the XML file.

As it is shown on figure 2, the XML file mainly contains the simulation name, a description of the simulation scheme with loops and breakpoints (*control* XML element) and a description of all the published data (*data* XML element). Scalar data and sequences (array) are precisely described with their type, their access permission and their location (on master process, replicated on all processes or distributed over all processes). The dimensions for the sequences must be detailed with its size, its offset and its its decomposition (block, cyclic, etc.) in the distributed case. Moreover, the XML `group` elements allow the user to logically associate different data (e.g. the *FluidBox mesh* group).

3.2. Instrumentation

As we already said, the integration of an existing simulation is done through source code annotations by a few function calls. Figure 3 presents the instrumented pseudo-code of *FluidBox* with the three classical phases, the initialization, the simulation loop and the ending phase. The initialization of all the *EPSN* infrastructure is simply done by calling the `init` function with the XML file name as argument. When considering parallel simulations, each process must also indicate its rank and the number of processes involved in the simulation. Then, each data described in the XML has to be pointed in the process memory and published (`publish`). Within the simulation body itself, one has to mark the loop evolution (`iterate`) and to locate the breakpoints (`barrier`) defined in the

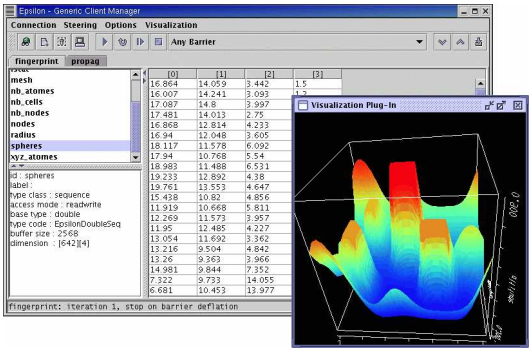

Figure 4. *EPSN* generic client.

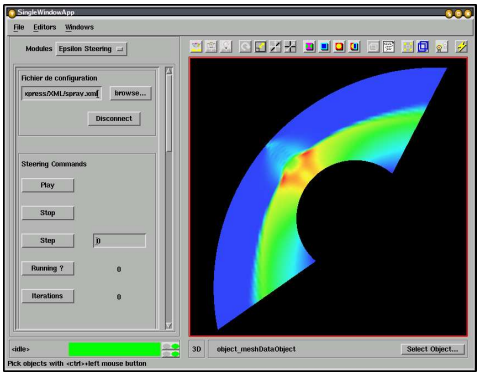

Figure 5. Semi-generic client (*FluidBox*).

XML file. One has also to place the data access areas (`lock/unlock`) and explicitly signal new data releases (`release`). In the example, only the energy field is modified during the iteration, so the other mesh components remain accessible during the whole computation. At the end of the process, the *EPSN* `exit` call properly terminates all the infrastructure. After that, if a client tries to connect or to send a request to the simulation, *EPSN* call gets a CORBA exception and returns an error status.

Eventually, when one runs an instrumented simulation, it starts the *EPSN* CORBA server, accessible through the CORBA naming service and waiting for client requests.

3.3. Visualization and Steering System

EPSN proposes three different strategies to construct a remote steering system. One could implement directly a steering system with the CORBA interface (IDL) but it is more convenient to use *EPSN* client API. The functionalities (see section 2.3) of this API allow the user to build a *specific client* precisely adapted to its simulation. One can also use the *EPSN generic client*, implemented in Java/Swing (Fig. 4). This tool can control and access the data of any *EPSN* simulation. Data are presented through simple numerical datasheets or can use basic visualization plug-ins. An intermediate way consists in implementing *semi-generic clients* using generic *EPSN* modules dedicated to the control, the data access and the visualization of complex objects (unstructured meshes, molecules, regular grids). This approach suits well with visualization environments (e.g. *AVS/Express* (Fig. 5)).

4. Preliminary Results

We have implemented a prototype of the *EPSN* platform, called *Epsilon*. This prototype is written in C++ and is based on *omniORB4* (*http://omniorb.sourceforge.net*), an high performance implementation of CORBA, and the associated thread library (*omniThread*). The results of this section come from experiments realized on two PC (Pentium IV) linked by a fast-Ethernet network ($100Mbps$).

The figure 6 presents the mean time needed by *Epsilon* to send different data sizes (from 1Kb to 4Mb) to both a local and remote client at each iteration (a `getp` request), whithout any computation performed. Remote *Epsilon* sendings are compared with TCP/IP communications upon which omniORB communicates and shows that *Epsilon* performances on data transfers are quite as good as TCP/IP's. The figure 7 presents the same experiment as previous, except that $80ms$ of computations are performed per iteration, half of time being in a unlocked access area. These results are compared with the computation time added to the TCP/IP sending time of the data. This figure demonstrates the overlapping capabilities of a fully asynchronous approach, as it is performed in *Epsilon*. Moreover, remote *Epsilon* performance is still under the sum of the computation time and the TCP/IP sending time. So, the *Epsilon* steering overhead is fully overlapped by the computation.

Finally, we have evaluated the instrumentation cost in the second experiment. It is quite negligible (less than 1%) and does not depend on clients for they are clearly disconnected from the simulation.

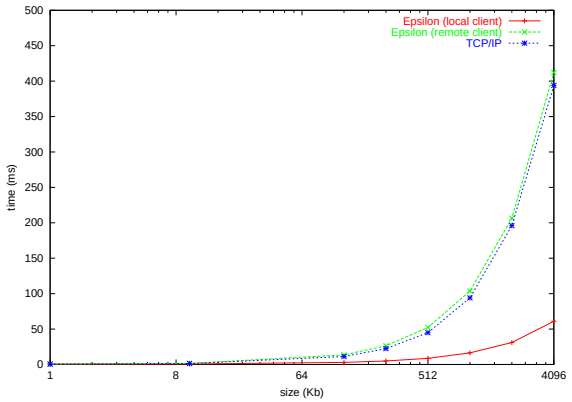

Figure 6. *Epsilon* communication benchmark.

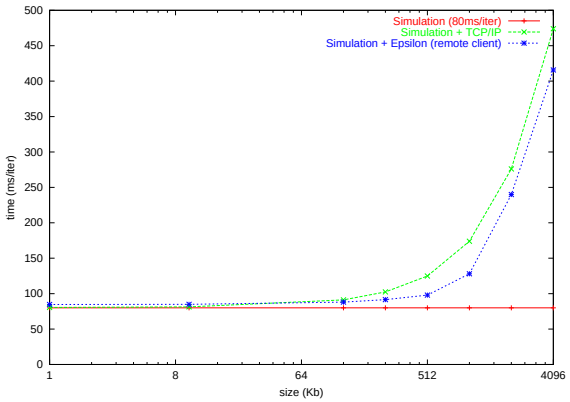

Figure 7. Data extraction from a simulation.

5. Conclusion and Prospects

As shown in this paper, *EPSN* architecture intends to provide a flexible and dynamic approach of computational steering. It proposes an instrumentation of existing simulations at a low cost and greatly capitalizes on CORBA features. Moreover, the parallel CORBA objects provides a suitable solution for most SPMD cases (with regular data distributions). The prototype *Epsilon* reveals itself as a really light and easy to use steering platform providing great capabilities of interaction. *Epsilon* validates the model of integration based on both a source code annotation and a XML description. It also proves that CORBA features present a great interest in the steering of applications with good performances. The developments of *EPSN* are now oriented on the integration of parallel and distributed simulations with irregular data distributions.

REFERENCES

[1] Jurriaan D. Mulder, Jarke J. van Wijk, and Robert van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1):119–129, 1999.

[2] S.G. Parker, M. Miller, C. Hansen, and C.R. Johnson. An integrated problem solving environment: the SCIRun computational steering system. In *Hawaii International Conference of System Sciences*, pages 147–156, 1998.

[3] Luc Renambot, Henri E. Bal, Desmond Germans, and Hans J. W. Spoelder. CAVEStudy: An infrastructure for computational steering and measuring in virtual reality environments. *Cluster Computing*, 4(1):79–87, 2001.

[4] J. A. Kohl and P. M. Papadopoulos. CUMULVS : Providing fault-tolerance, visualization, and steering of parallel applications. *Int. J. of Supercomputer Applications and High Performance Computing*, pages 224–235, 1997.

[5] S. Rathmayer and M. Lenke. A tool for on-line visualization and interactive steering of parallel hpc applications. In *Proceedings of the 11th IPPS'97*, pages 181–186, 1997.

[6] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.

[7] Alexandre Denis, Christian Pérez, and Thierry Priol. Towards high performance CORBA and MPI middlewares for grid computing. In *Proceedings of 2nd IWGC*, pages 14–25, 2001.

[8] Alexandre Denis, Christian Pérez, and Thierry Priol. Portable parallel CORBA objects: an approach to combine parallel and distributed programming for grid computing. In *Proc. of the 7th Intl. Euro-Par'01 Conference (EuroPar'01)*, pages 835–844, 2001.

[9] B. Nkonga and P. Charrier. Generalized parcel method for dispersed spray and message passing strategy on unstructured meshes. *Parallel Computing*, 28:369–398, 2002.