Toward a Computational Steering Environment for Legacy Coupled Simulations

Nicolas Richart Aurélien Esnard Olivier Coulaud
Projet ScAlApplix, INRIA Futurs and LaBRI UMR CNRS 5800,
351, cours de la Libération, F-33405 Talence, France
Email: {richart, esnard, coulaud} at labri.fr

Abstract

In this paper, we present an abstract model to steer legacy coupled simulations that follow the Multiple-SPMD paradigm or the client/server paradigm. This model extends our previous work for the steering of classical SPMD simulations. It describes the application in terms of execution flow, complex distributed data objects and user interactions. Thanks to this abstraction, we define a coordination algorithm that allows us to efficiently interact with the simulation and to overcome the time-coherence problem raised by coupled 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 distributed codes running on heterogeneous networks of parallel computers (e.g. multi-physics simulations). These simulations are HPC software that have a high degree of internal complexity. For years, the scientific community has expressed the need for new computational steering tools in order to better grasp the complex internal structure of large-scale scientific applications.

A computational steering environment is defined in [15] as a communication infrastructure, coupling a simulation with a remote user interface, called steering system or steering client. This interface usually provides on-line visualization and user interaction. Over the last decades, many steering environments have been developed; they distinguish themselves by some critical features such as the simulation representation or the interaction mechanism used to ensure the simulation coherence. Magellan [20] represents a simulation as a simple collection of instrumentation points. To notify the time evolution, a timestamp is explicitly specified by the end-user on each instrumentation point but nothing ensures the interaction coherence over the parallel processes. CUMULVS [12] considers parallel simulations as single-loop programs with a unique instrumentation point. Thanks to this simplification, it implements synchronization mechanisms relying on the internal loop counter and provides time-coherent accesses to distributed data. VASE [9] proposes a high-level model for the representation of simulations based on a control-flow graph (CFG) and provides a data-flow coupling model. However, this environment is only intended for sequential simulations. All these environments are representative of the state-of-the-art: however efforts have to be done to steer more complex simulations that are not limited to the classic SPMD execution paradigm.

We have developed in previous works, the *EPSN* framework [4], that introduces an abstract representation of parallel SPMD [6] simulations in order to capture its execution flow and to allow safe steering interactions. With few annotations in the source code with the *EPSN* API, we can steer interactively a legacy parallel simulation. A steering client can connect to the simulation, perform different kinds of steering requests, and finally disconnect. We distinguish three types: the *control requests* which allow to control the execution flow (play, step, pause) of the simulation, the *data requests* which allow to access data values (get, put), and the *user-defined interactions* which are functions of the simulation that can be remotely invoked. In our case the steering is driven by the client requests. In this way, if the simulation doesn't receive any request, it is not disturbed by the steering framework. The execution of the interaction request is defined by an abstract representation of the simulation. Up to now this abstraction can only represent parallel SPMD simulations. But modern scientific simulations to model complex phenomena like earth system modeling, biology, solid-fluid interactions are typically made up of several parallel codes interconnected together. So a natural evolution for our steering environment is to manage this type of simulations.

Such coupled simulations are more and more common because the complexity of the multiphysics or multiscale models can not anymore be handled by simple and monolithic parallel codes. Those simulations can be divided in two main classes. One approach couples different SPMD

codes, leading to Multiple-SPMD code (M-SPMD). Such codes are typically characterised by an explicit time loop, that describes the evolution of the whole simulation. A second approach considers codes built upon the client/server paradigm where the servers or/and the client could be parallel, as in the case of GRID RPC [18] applications. In this approach, only the client has an actual time loop, and it performs remote procedure calls on different servers.

In domains like meteorology or solid-fluid interactions, we found lots of M-SPMD codes that often rely on dedicated high-level coupling frameworks like ESMF [8], Prism [19], or more generic coupling frameworks like PCI [3], or MCT [10]. There are also simulations coupled by ad-hoc solutions based on communication frameworks, which handle data redistribution like MpCCI [11], InterComm [13], or RedGRID [17]. These coupled simulations can also be based on Grid infrastructure: in this case, they are built over framework such as Cactus [1], CCA [14], or RealityGrid [16]. These frameworks offer some light steering functionalities such as the control of execution flow or simple monitoring capabilities. However, most of these solutions does not propose a sufficient abstraction to allow the steering of the simulation.

In this paper, we extend the model we proposed in [6], based on a hierarchical task representation. This new model allows to capture the complexity of coupled simulations, hence giving us the means to safely steer them. We first introduce the notion used in our previous model to represent parallel SPMD simulations. Then we propose an extension to this model for M-SPMD and client/server paradigms. Finally, we valid the capability of the proposed model to steer M-SPMD simulations on a multiscale application, with a prototype based on the current *EPSN* implementation.

2. Model for the Steering of Parallel SPMD Simulations

In order to efficiently steer parallel simulations, we have introduced in [6] a high-level description model, that provides a uniform view of a parallel simulation in terms of execution flow, data and user actions. This model requires to annotate the source code with the *EPSN* API [7] (instrumentation phase). It is divided in three parts: a description of the data (distribution, storage) that the user want to access, a description of the structure of the simulation to capture the execution flow, and a description of the possible steering interactions (control, data access, user-defined actions). All the high-level description information are provided by a XML file associated with a simulation. It allows us to build a light API for the source-code annotation. In the following subsection we will present in more detail the different parts of the model.

2.1. Abstract Model

Data description. Not all the data in a simulation are relevant from the steering point of view. So, during the instrumentation phase, one must select the data that the client want to interact with. This can be quite complex because, in order to access to the data in parallel codes, we have to describe how the data are spread between all processes (distribution) and how they are mapped in the memory of each process (storage). To do that, we have introduced a generic data model [17] that allows to describe complex objects such as structured grids, unstructured meshes, particle sets and scalar parameters. In a complex object, one must specify several variables, called *series*, like pressure, velocity, etc. The different data series within a complex object share some common information, such as distribution or access permissions.

Simulation description model. The structure of a SPMD simulation is described by means of a Hierarchical Task Model (HTM) [6]. The HTM is a task tree that describes only the tasks that we want to steer in the simulation. We distinguish three kinds of task (Fig. 1(a-c)): *basic task* for logical block of code, *conditional task* for conditional structure (if-then-else, switch-case) and *loop task* for iterative structure (for, while). A task is simply delimited by two instrumentation points with the same *ID* in the code: the *begin* and the *end* point. These points are precisely located in the source code by calls to the *EPSN* API. In addition, we consider a degenerated task, called the *point task* (Fig. 1d) where the begin point and the end point are merged in a single point. All tasks (except point task) can contain other sub-tasks and the entire simulation is represented by a simple task that contains all the hierarchy as shown in figure 2.

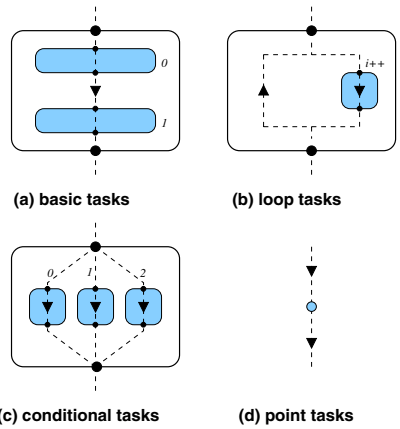

Figure 1. Different kinds of task (blue zones) and instrumentation points (black dots).

User-defined interactions. In addition to the control and the data request, we consider another kind of interactions called *user-defined interaction*. It associates a callback function in the simulation code with an interaction *ID* in the XML file. Thus, this function can be called by a remote steering client.

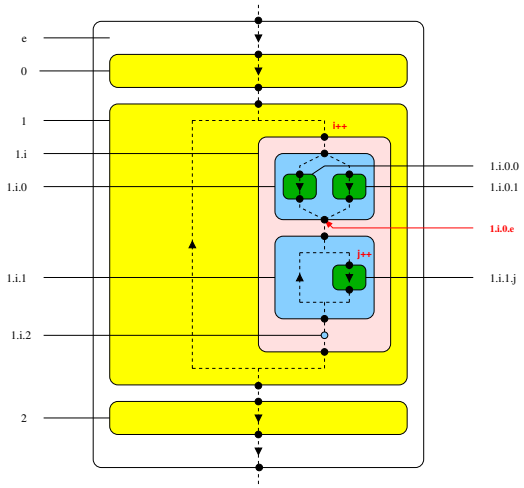

Figure 2. Example of a basic HTM with the corresponding dates.

2.2. Coherence Solution

The steering of parallel simulations come up against a serious coherence problem. Indeed, data distributed over processes must be accessed carefully to ensure they are presented to the visualization system in a meaningful way. In other words, the problem is to guarantee, that a steering request is executed at the same time by all the processes. So, in addition to the HTM, we have introduced a date system, and notion of *task context* that precises on which task an interaction is allowed. Finally, we present an algorithm that determines when a request will be treated on each process.

Date system. It is based on the notion of *point date* defined as a n-tuples $(d_0, d_1, ..., d_{n-1})$ in $\mathbb{N}^{n-1} \times \{b, e\}$, noted $d = d_0 \cdot d_1 \cdot \ldots \cdot d_{n-1}$. The index i for $i < n-1$, refers to the depth in the task hierarchy and the value d_i is the number of the task at this depth as illustrated in figure 2. The last element corresponds to a point information: its value is *b* for a begin point and *e* for an end point. For example, the date $1.i.0.e$ in figure 2 means that we are at the end of the first task (depth 3) at the $(i+1)^{th}$ iteration (depth 2) of the second task (depth 1). In [5], we have defined an strict order on these dates which reflects the order of apparition of instrumentation points during the execution.

Task context. In addition to the date system, we enrich the model with a context associated to each task. This context consists of information about possible interaction on this task. For user-defined interactions, we describe if the interaction is allowed or not during the task. For data, this context specifies if we can read or write the data in the task (*readable, writable*), if a new release of the data is produced (*modified*) or if the data is not accessible (*protected*).

Coordination algorithm . Thanks to the date system that allows to follow the execution flow, one can define a coordination algorithm to solve the coherence problem. Our algorithm schedules the treatment of requests and thus avoids strong synchronization between processes. The algorithm starts when each process locally receives a request. It is then divided in four main phases:

1. On each process, we freeze all instrumentation points which are now blocking. This is called the *freeze order*. Therefore, the simulation cannot run over the next task and stays at the same date.

2. It consists in determining the *current date* on each process. All current local dates which are not necessarily the same, are centralized on a particular node (called *proxy*).

3. The proxy determines the *scheduling date* that is defined as the greatest current date relatively to the order on point dates.

4. This date is then broadcasted to all nodes, and once it is received by the nodes, they release the freeze order.

Finally, the request will be executed in parallel by all processes at the first date after the scheduling date that satisfies a *local condition* that depends both on the request kind and the task contexts. The fact that the simulation follows a SPMD execution ensures that all processes will do their treatments at the same date.

2.3. Limitation of this Model

This model has been designed to represent SPMD simulations. We can also represent M-SPMD simulations if all SPMD codes have an identical HTM, but we reach the limit of the model when we want to represent more complicated distributed simulations. Indeed we cannot ensure the time coherence between the codes because they do not have a common representation, so they do not share the same date system. Furthermore, our model cannot manage data that are distributed over several parallel codes. To overcome those limitations, we will describe in the next section an extension to our model.

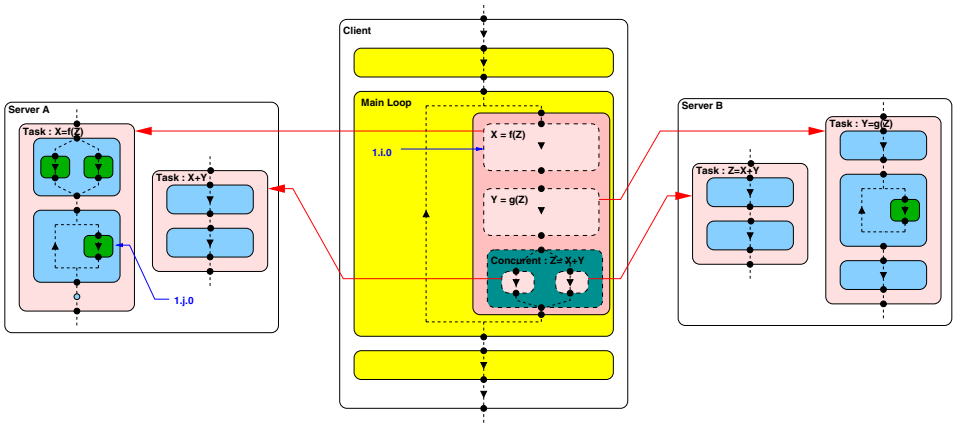

Figure 3. View of a distributed HTM with both remote and concurrent tasks.

3. New Model for the Steering of Parallel-Distributed Simulations

To steer parallel-distributed simulations such as M-SPMD and client-server simulations, we decide to extend our previous model in two ways. First, we enrich the description of the simulation structure to take into account more complex tasks; second we extend the description of data objects when they are defined across several codes. As a consequence, one must also adapt the coordination algorithm to provide the required coherence for the steering requests in the parallel-distributed case.

3.1. Simulation Description Model

The extended model is based on the HTM previously described. In order to capture the complexity of M-SPMD and client/server simulations, we introduce two new kinds of task: the *remote task* and the *concurrent task*. A *remote task* is a task that is not defined in the same code as the *main HTM* but in another HTM located on a remote code. This is illustrated in the figure 3 by the tasks named $X = f(Z)$, $Y = g(Z)$, and $Z = X + Y$ which are called by the client, but defined in *partial HTMs* for servers A and B. A *concurrent task* is a meta-task that is created to specify that the sub-tasks inside can be executed simultaneously on different codes. For instance, the tasks $Z = X + Y$ of the servers A and B in figure 3 are executed at the same time by the client. The XML file corresponding to this example is given in figures 4 and 5. All the dark blue keywords are the ones added in the new model.

In the case of a client/server simulation, our approach fits well because it is based on a client that realizes remote calls to one or several server codes. We describe the client with a classical HTM in which we add remote tasks that are defined on the servers. In the server codes, we define a set

```
<simulation id="client">
<htm>
 <task id="init" />
  <loop id="main-loop">
   <task id="body">
    <task id="X=f(Z)" remote="yes" location="ServerA"/>
    <task id="Y=g(Z)" remote="yes" location="ServerB"/>

    <concurrent-tasks>
     <task id="Z=X+Y" remote="yes" location="ServerA"/>
     <task id="Z=X+Y" remote="yes" location="ServerB"/>
    </concurrent-tasks>
   </task>
  </loop>
 <task id="finalize" />
</htm>
</simulation>
```

Figure 4. XML describing the main HTM of the figure 3.

```
<simulation id="ServerA">
 <function id="X=f(Z)">
  <switch id="switch1">
   <task id="switch-task1" />
   <task id="switch-task2" />
  </switch>
  <loop id="loop1">
   <task id="body" />
  </loop>
 </function>

 <function id="Z=X+Y">
  <task id="task1" />
  <task id="task2" />
 </function>
</simulation>
```

Figure 5. XML describing the Server A shown in figure 3.

of tasks in a partial HTM, that could be remotely called by the client code (cf. *function* keywords in the XML).

In a M-SPMD simulation the correspondence is not so obvious because all codes have an explicit loop and then we have potentially as many HTMs as codes. In this case, our approach consists in building a *virtual HTM* that represents the main loop and the "common" tasks shared between those codes. The virtual HTM acts as the main HTM except there is no code associated with. The concurrent tasks are defined as "common" tasks that contain sub-tasks which grasp the fine-grain differences between codes. On the codes, we have only to describe the sub-tasks in a partial HTM as for a server. The aim of the virtual HTM construction is to provide a common representation for client/server and M-SPMD simulations.

3.2. Data Description

```
<simulation id="client">
 <data>
  <meta-object id="D_dist">
   <serie id="s1" data_id="D_a" serie_id="s2"
          function="X=f(Z)" task_id="switch1"
          location="ServerA" />
   <serie id="s2" data_id="D_b" serie_id="s1"
          function="Y=g(Z)" task_id="finalize"
          location="ServerB" />
  </meta-object>
 </data>
</simulation>
```

Figure 6. Description of the meta-object.

```
<simulation id="ServerA">
 <data>
  <mesh id="D_a">
   <serie id="s1" />
   <serie id="s2" />
  </mesh>
 </data>
</simulation>
```

Figure 7. Description of data of Server A.

When we visualize online results of a distributed simulation, we may want to see data that are for example, on two codes: so we need to have a precise description of data to get a coherent version from these two coupled codes. Indeed when we want to access such a data object, we have to know in which code it is mapped, and when it is coherent. To do that, we define a meta-object that rely on the existent data description. This meta-object is composed by several series from different objects that may be on different codes. For all series, one must also give the task in which it is coherent with other variables of the meta-object. For example in the simulation described in figure 3, we consider a data object D_A with two variables $D_A(s_1)$ and $D_A(s_2)$ in the server A, and a data object D_B with one variable $D_B(s_1)$ in the server B. We define D_{dist} as the meta-object composed by $D_A(s_2)$ in the first sub-task of the task $X = f(Z)$ and $D_B(s_1)$ in the last sub-task of $Y = g(Z)$. An illustration of this example is given by the XML in figure 6 and 7. With such information, we know precisely when we can access the data in order to have a global coherence for the meta-object.

3.3. Coordination Algorithm

In order to define the new coordination algorithm, one must first extend the date system to take into account the extra complexity introduced by the new classes of task. The *global current date* of a task, is defined as the current date in the main HTM (*common current date* concatenated with *local current date* in the partial HTM). For example, if we are in the task $X = f(Z)$ of the server A (Fig. 3), the date in the main HTM is something like $1.i.0$ and the date on the server A is something like $1.j.0$; hence the global current date is $1.i.0.1.j.0$. For concurrent tasks, one must consider different dates for each task, but one cannot compare those dates for it has no sense.

The new coordination algorithm is roughly the same as the one described for SPMD simulations. But here we come up against new difficulties. Firstly the dates are not as easily determined as before, particularly for remote tasks. Secondly the meta-objects cannot be easily accessed, because they could be shared between different objects and different codes. We distinguish two types of coordination, one for control requests and the other for data requests that will be presented in the next sub-sections.

As in the SPMD model, we introduce a unique access point for each parallel code and a *main proxy* for the whole coupled simulation as shown in figure 8. The main proxy which is the unique access to the coupled simulation, has the knowledge of the main loop (main HTM), and sub-proxies only know the local tasks (partial HTMs).

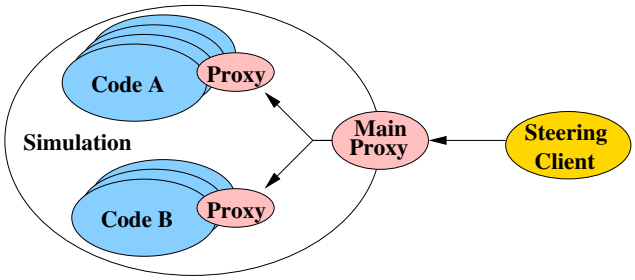

Figure 8. A logical view of a parallel-distributed simulation in the EPSN framework.

Control Request. This algorithm is used for the pause request. In order to ensure the time coherence for this request, we limit the pause to the only points of the main HTM. Our algorithm is divided in three steps:

1. The steering client sends the control request to the main proxy. Then the proxy broadcasts a freeze order to all the codes through their sub-proxies.

2. The main proxy determines the current common date restricted to the main HTM. In the M-SPMD case, this is the date of the next point reached on the main HTM. In the client/server case, we determine this date with the SPMD algorithm on the client code that could be parallel (Sec. 2.2).

3. The point corresponding to the scheduling date is set to be blocking.

The play request simply consists to release the blocking points, so it does not require any coordination. In the case of a non-concurrent remote task, one can improve our algorithm by finding a coherent pause point in this task.

Data Request. In this case, it is more complicated as for control request, because we must consider the information about coherence given by the meta-objects. We have divided our algorithm in six steps as follows:

1. The steering client sends the data request to the main proxy. Then the proxy broadcasts a freeze order to all the codes through their sub-proxies.

2. The main proxy determines the current common date restricted to the main HTM as in the previous algorithm.

3. The main proxy sends the data request and the current common date to all processes that have a piece of the requested data, through their sub-proxies.

4. When receiving the requests, the processes determine their local current date with the previous SPMD algorithm.

5. The sub-proxies concatenate the current common date with their current local date to determine the scheduling date that is local to all codes.

6. The request is executed after the scheduling date on each code, at the first date that fulfills the local condition enriched by the meta-object.

These algorithms have been integrated in the *EPSN* framework as a preliminary prototype in order to experiment the different aspects of the model.

4. Example

In order to validate our approach, we will see how the M-SPMD code called LibMultiScale [2] is steered by our model. LibMultiScale simulates crack propagation by a multi-scale approach coupling an atomistic model and a continuum model. The material is described at the atomistic level by a crystal of atoms and its evolution is given by molecular dynamics, while at the continuum level the material is described by the elastodynamics model which is discretized by the finite element method. The two codes, molecular dynamics and elastodynamics, are parallel and follow the SPMD execution model.

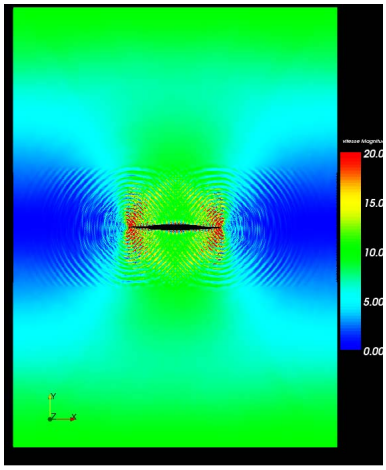

Figure 9. Online visualisation of a LibMultiScale 2D case study.

On the domain where the two different scales are coupled, a set of constraints is applied to ensure that the displacements of the two models are the same. Then, a set of algebraic differential equations is obtained, which are solved by the SHAKE algorithm on each code.

A step of the time integration is decomposed in two main tasks as follows:

1. Leapfrog
   - $V^{n+\frac{1}{2}} = V^n + \frac{\Delta t}{2} F^n$
   - $X^{n+1} = X^n + \Delta t V^{n+\frac{1}{2}}$
   - evaluate the force $F^{n+1} = F(X^{n+1})$
   - $V^* = V^{n+\frac{1}{2}} + \frac{\Delta t}{2} F^{n+1}$
2. Constraint
   - solve the constraint problems $G(V^{n+1} + \lambda \frac{\partial G}{\partial X}) = 0$
   - update the velocity

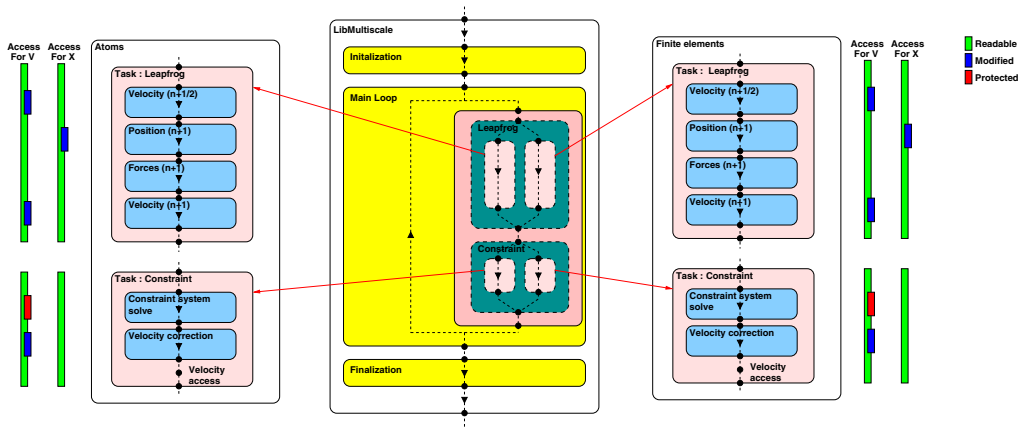

Figure 10. Abstract view of LibMultiscale in our extended model.

where X^n (resp. V^n) is the atomistic or FE displacements (resp. its velocity) at the time t_n, F^n the force applied on the system and G the constraint.

To capture cleverly the flow of this algorithm one consider two main tasks: the Leapfrog and the constraint task. The first task is decomposed in four sub-tasks to follow the different steps of this algorithm. The second task is then decomposed in two sub-tasks. As shown in figure 10, we introduce two concurrent tasks corresponding to the main tasks in the time loop. In that particular example, the high-level HTM description is the same for each code.

The data that are important to extract are the displacement and the velocity. In the atoms code, we define the complex object *Atoms* (see figure 11) in which there are the initial position X^0 (at time $t = 0$), the current position X, and the velocity V. The distribution of the atoms on the processes is based on the position series X. The displacement is defined by $X - X^0$

```
<simulation id="Atoms">
 <data>
  <points id="Atoms">
   <serie id="X0" />
   <serie id="X" />
   <serie id="V" />
  </points>
 </data>
</simulation>
```

Figure 11. Description of the Atoms complex object.

In the finite element code, we define the complex object *FE* (see figure 12). This object is distributed over the processes by a mesh partitioning. We consider the following series in the object: vertex position, velocity (located on the vertex) and stress (located on the mesh element).

For online visualization (Fig. 9), we are interested to see

```
<simulation id="FiniteElements">
 <data>
  <mesh id="FE">
   <serie id="vertex" />
   <serie id="connectivity" />
   <serie id="displacement" />
   <serie id="velocity" />
   <serie id="stress" />
  </mesh>
 </data>
</simulation>
```

Figure 12. Description of the FE complex object.

the displacement in all the material, this means that we want to extract from the simulation the series of position X^0 and X in the object *Atoms* located on code *Atoms* and the series displacement in the *FE* object located on code *Finite Elements*. To do that we construct the meta-object shown in figure 13. For the velocity meta-object, we have a similar description as for the displacement, except for the task context as shown in figure 10.

In meta-objects, one must define for each data series a task where the data will be globally coherent with other series. The choice of this task depends on the user needs. For instance, an end-user only wants to visualize the final result (the corrected velocities), so he will access the velocity series at task *Velocity access*. In the case of a developer who needs to evaluate the impact of the correction, he wants to see the velocities before and after the correction step.

5. Conclusion

In this paper, we propose a new steering model that addresses modern scientific simulations such as the multiscale simulations presented in the final example. This model

```
<simulation id="LibMultiScale">
 <data>
  <meta-object id="displacement">
   <serie id="atoms-displacement" data_id="Atoms"
          serie_id="X"
          function="Leapfrog" task_id="Forces(n+1)"
          location="Atoms" />
   <serie id="fe-displacement" data_id="FE"
          serie_id="displacement"
          function="Leapfrog" task_id="Forces(n+1)"
          location="FiniteElement" />
  </meta-object>
 </data>
</simulation>
```

Figure 13. Description of the meta-object displacements.

introduces a common abstraction for both M-SPMD and client/server applications. Thanks to this abstraction, we define a new coordination algorithm that allows to perform time-coherent interactions over the whole coupled simulation. Our prototype is currently available at INRIA Gforge web site as a private project for internal members and will be soon publicly accessible.

Acknowledgments

This works has been supported by the ANR program (grant number ANR-05-MMSA-0008-03) and by the Conseil Regional d'Aquitaine (PhD. grant). The authors also would like to thank G. Anciaux for the LibMultiScale code used in this article.

References

[1] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In *Proceedings of Supercomputing 2001*, Denver, USA, 2001.

[2] G. Anciaux, O. Coulaud, and J. Roman. High Performance Multiscale Simulation for Crack Propagation. In *Procceedings of the 8th Workshop on High Performance Scientific and Engineering Computing*, Columbus, Ohio, USA, August 2006.

[3] T. Bulatewicz, J. Cuny, and M. Warman. The potential coupling interface: metadata for model coupling. In *Proceedings of the 2004 Winter Simulation Conference*, volume 1, pages 175–182, Washington D.C., USA, 2004.

[4] EPSN. A Computational Steering Environment for Distributed Numerical Simulations.

[5] A. Esnard. *Analyse, conception et réalisation d'un environnement pour le pilotage et la visualisation en ligne de simulations numériques parallèles*. Informatique, Université de Bordeaux 1, décembre 2005.

[6] A. Esnard, M. Dussere, and O. Coulaud. A Time-Coherent Model for the Steering of Parallel Simulations. In *Euro-Par 2004 Parallel Processing*, pages 90–97, Pisa, Italy, 2004. Springer-Verlag.

[7] A. Esnard, N. Richart, and O. Coulaud. A Steering Environment for Online Parallel Visualization of Legacy Parallel Simulations. In *Proceedings of the 10th International Symposium on Distributed Simulation and Real-Time Applications (DS-RT 2006)*, pages 7–14, Torremolinos, Malaga, Spain, October 2006. IEEE Press.

[8] C. Hill, C. DeLuca, V. Balaji, M. Suarez, and A. da Silva. Architecture of the Earth System Modeling Framework. *Computing in Science and Engineering*, 6(1), 2004.

[9] D. Jablonowski, J. Bruner, B. Bliss, and R. Haber. VASE: The Visualization and Application Steering Environment. In *Proceedings of Supercomputing'93*, pages 560–569, 1993.

[10] E. O. Jay Larson, Robert Jacob. The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models. *International Journal of High Performance Computing Applications*, 19(3):277–292, 2005.

[11] W. Joppich and M. Kürschner. Mpcci-a tool for the simulation of coupled applications. *Concurr. Comput. : Pract. Exper.*, 18(2):183–192, 2006.

[12] J. Kohl, P. Papadopoulos, and G. Geist. CUMULVS: Collaborative Infrastructure for Developing Distributed Simulations. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, Mars 1997.

[13] J.-Y. Lee and A. Sussman. Efficient Communication Between Parallel Programs with InterComm. Technical Report CS-TR-4557 and UMIACS-TR-2004-0, University of Maryland, Department of Computer Science and UMIACS, January 2004.

[14] S. Lefantzi, J. Ray, and H. N. Najm. Using the Common Component Architecture to Design High Performance Scientific Simulation Codes. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 2003.

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

[16] RealityGrid. Moving the bottleneck out of the hardware and back into the human mind.

[17] RedGRID. Parallel Data Redistribution Library. http://www.labri.fr/ esnard/Research/RedGRID.

[18] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. GridRPC: A Remote Procedure Call API for Grid Computing, june 2002.

[19] S. Valcke, D. Declat, R. Redler, H. Ritzdorf, T. Schoenemeyer, and R. Vogelsang. The PRISM Coupling and I/O System. In *VECPAR'04, Proceedings of the 6th International Meeting*, volume VOL. 1 : High performance computing for computational science, Universidad Politecnica de Valencia, Valencia, Spain., 2004.

[20] J. Vetter. Experiences using computational steering on existing scientific applications. In *Ninth SIAM Conf. Parallel Processing*, 1999.