Vers le pilotage des simulations numériques sur la grille

Michaël Dussere, Aurélien Esnard

RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Vers le pilotage des simulations numériques sur la grille

Michael Dussere, Aurélien Esnard

Projet ScAlApplix, INRIA Futurs et LaBRI UMR CNRS 5800,
351, cours de la Libération, F-33405 Talence - France
{dussere,esnard}@labri.fr

Résumé
Le projet *EPSN* [1] a pour but de développer un environnement logiciel pour le pilotage de simulations numériques parallèles et distribuées. L'environnement *EPSN* permet d'instrumenter les simulations numériques et de construire des applications clientes, typiquement dédiées à la visualisation, capables de contrôler, d'extraire ou de modifier les données de la simulation à distance. Afin de mettre en oeuvre un couplage générique, nous avons introduit un modèle abstrait des simulations interactives reposant sur une description XML et une annotation du code source. Cette abstraction nous permet de développer des clients de pilotage indépendamment d'une simulation particulière. La couche de communication d'*EPSN* met à profit la technologie CORBA (portabilité, interopérabilité et transparence du réseau), tout en masquant aux utilisateurs finaux la complexité inhérente de cet intergiciel. Cet article présente le prototype séquentiel *Epsilon* ainsi que les premiers travaux autour des simulations parallèles SPMD.

Mots-clés : simulation numérique, visualisation scientifique, couplage de code, interaction, CORBA.

1. Introduction

Grâce à la constante évolution des moyens de calculs, les simulations numériques deviennent de plus en plus complexes. Il n'est pas rare de trouver différents modèles et codes couplés sur un réseau hétérogène (e.g. les simulations multi-physiques). Même si les enjeux de la simulation interactive sont aujourd'hui bien perçus, la communauté du calcul scientifique exprime toujours le besoin d'une nouvelle génération d'outils pour le pilotage des simulations numériques sur des environnements distribués comme la grille. Le domaine de la simulation interactive ou *computational steering* a pour but d'améliorer le processus de simulation numérique (modélisation, calcul, analyse) en le rendant plus interactif. Dans cette approche, l'utilisateur n'attend plus passivement les résultats de la simulation, mais il visualise "en temps-réel" l'évolution des données calculées et peut interagir tout au long de la simulation en pilotant les algorithmes ou en modifiant certains paramètres à la volée. Un tel outil peut s'avérer très utile aux scientifiques pour la compréhension des phénomènes modélisés et la détection d'erreurs dans le cas des simulations longues.
Un environnement de pilotage est défini dans [1] comme une infrastructure de communication couplant une simulation avec une interface utilisateur, appelé système de pilotage, qui offre classiquement des moyens de visualisation et d'interaction avec la simulation. Au cours des dix dernières années, beaucoup de projets ont abordé la problématique de la simulation interactive. Les environnements logiciels résultant de ces travaux se distinguent sur un ensemble de points critiques tels que le modèle d'intégration d'une simulation, l'infrastructure de communication ou le système de pilotage. Un premier modèle d'intégration est l'approche PSE (Problem Solving Environment) adoptée par SCIRun [2] et COVISE [3]. Dans cette approche, l'utilisateur construit une nouvelle simulation en assemblant des composants de calcul, de visualisation et d'interaction selon un modèle de programmation visuelle orienté flux de données. A l'opposé, CAVEStudy [4] interagit uniquement avec une application en exploitant ses entrées/sorties standards. Toutefois, la majorité des environnements de pilotage, comme ceux que

[1] *EPSN*: Environnement pour le Pilotage de Simulations Numériques (projet ACI-GRID PPL02-03).

nous citons par la suite, utilise un modèle d'intégration basé sur l'instrumentation du code source. Nous avons choisi cette approche car elle permet de définir un couplage, précis et performant, adapté au code de la simulation. Concernant l'infrastructure de communication, divers problèmes apparaissent notamment pour des simulations parallèles (SPMD) et distribuées (MPMD) comme le transfert hétérogène des données, les protocoles de communication, les mécanismes de synchronisation ou le problème de redistribution des données. Dans VIPER [5], l'infrastructure de communication repose sur des appels de procédures distantes (RPC) et sur XDR pour l'encodage des données. Magellan [6] et Falcon [7] utilisent un système d'évènements construit au dessus de DataExchange [8] prenant en charge la communication dans un environnement hétérogène. CUMULVS [9] utilise une infrastructure de communication de haut-niveau basée sur PVM qui permet d'extraire des données distribuées régulièrement à la manière de HPF. Dans le projet *EPSN*, nous nous proposons d'explorer les possibilités de la technologie CORBA pour définir une infrastructure de communication originale permettant un déploiement simple sur les systèmes très hétérogènes comme les grilles de calcul. Ce choix est renforcée par l'émergence récente des *objets parallèles CORBA* qui permettent l'envoi de requêtes entre applications parallèles.
Dans ce papier, nous décrivons tout d'abord les bases de l'environnement *EPSN*. Nous présentons ensuite le prototype séquentiel *Epsilon* et son extension vers le parallélisme. Finalement, nous détaillons les étapes nécessaires au développement d'une simulation interactive en illustrant notre propos par un exemple.

2. L'environnement *EPSN*

L'environnement *EPSN* a pour objectif de piloter les simulations numériques impliquant un processus de calcul itératif, comme par exemple les schémas de résolution en temps. Afin d'être aussi générique que possible, nous introduisons un modèle abstrait des simulations interactives. Ce modèle décrit les éléments impliqués dans le processus de pilotage. L'utilisateur complète cette description en annotant le source de la simulation avec une API. Cette abtraction permet de développer, à l'aide d'une seconde API, des clients de pilotage indépendants d'une simulation particulière.

2.1. Principe de l'instrumentation

Nous considérons les simulations comme des processus itératifs impliquant un ensemble de données et une hiérarchie de boucles de calcul modifiant ces données. Chaque boucle est associée à un compteur et l'ensemble de ces compteurs permet de suivre précisément l'évolution de la simulation. Le modèle comprend, en outre, la description de points d'arrêt identifiant des états stables de la simulation. Ces points pourront être activés ou désactivés à distance pour contrôler le déroulement de la simulation. Souvent, l'identification de la seule boucle principale suffit au contrôle global de la simulation.
Au cours des calculs, les données passent successivement d'un état cohérent à un état incohérent. Les interactions entre le client et la simulation doivent concerner exclusivement des données cohérentes pour qu'elles soient interprétées correctement par le client mais aussi pour ne pas altérer le déroulement de la simulation. De plus, il est souvent intéressant de corréler certaines données et de les interpréter dans leur ensemble, ce qui amène à définir une notion de groupe ainsi qu'une notion de version des données. Dans ce contexte, l'utilisateur doit faire apparaître dans la simulation les plages d'accessibilité aux données ainsi que l'évolution des données.
Le principe d'*EPSN* consiste à établir le pilotage d'une simulation numérique sur la base d'une représentation abstraite de cette simulation. Cette représentation est obtenue en exhibant les informations clés de la simulation. Dans un premier temps, l'utilisateur doit décrire dans un fichier XML les données ainsi que les boucles et les points d'arrêt qu'il souhaite exporter. Chacun de ces éléments est associé avec un identifiant unique. L'utilisateur doit alors annoter son code source à l'aide des fonctions de l'API pour construire une simulation pilotable. Cette phase d'annotation constitue la seule modification à apporter au code source. Notons que l'instrumentation par appel de fonctions, nous permet d'aborder la plupart des simulations numériques qui sont, le plus souvent, des applications itératives écrites en C/C++ ou Fortran.

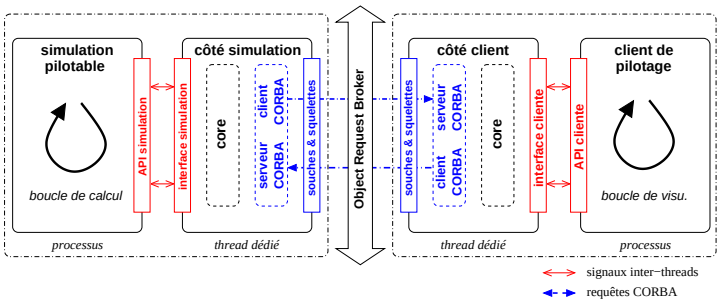


Figure 1: Infrastructure de communication d'*EPSN*.

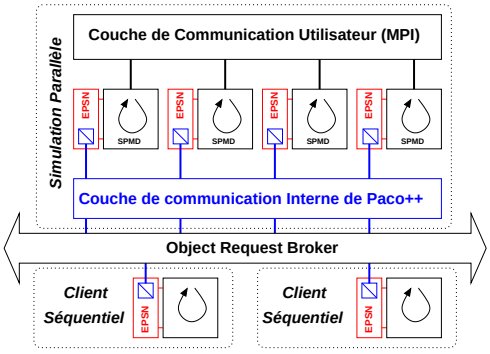


Figure 2: Extension avec PaCO++.

2.2. Architecture et infrastructure de communication

EPSN présente une infrastructure à la fois dynamique et distribuée. Cette infrastructure est basée sur une relation client/serveur entre le système de pilotage qui tient le rôle du client et la simulation qui s'apparente à un serveur. Les clients ne sont pas connectés aux serveurs : ils utilisent un mécanisme de requêtes asynchrones et concurrentes pour consulter à distance les données des simulations. Ces caractéristiques font d'*EPSN* un environnement flexible et dynamique. Elles permettent à la fois aux simulations de recevoir des requêtes de plusieurs clients de pilotage (multi-clients) et à ces clients d'accéder à différentes simulations (multi-applications).

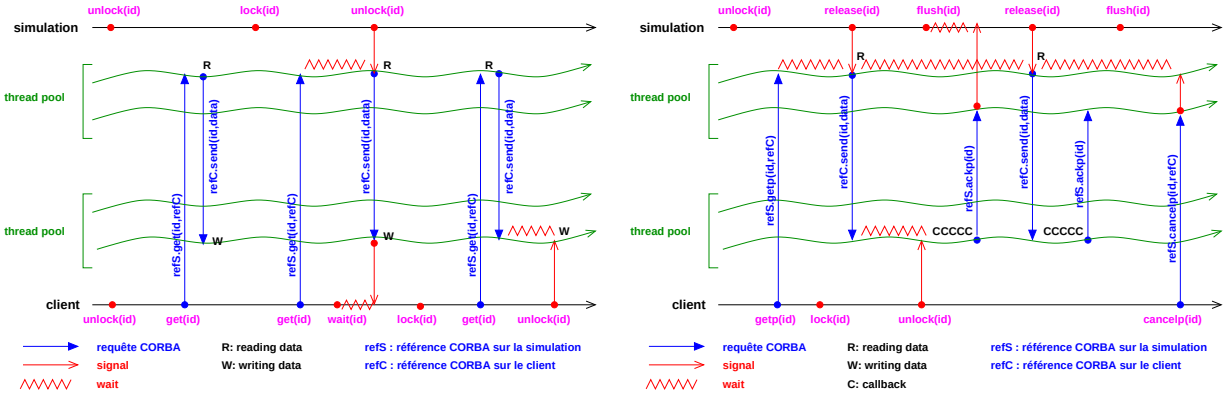
D'autre part, *EPSN* introduit peu de perturbations dans le déroulement des simulations. En effet, toutes les requêtes, de contrôle ou d'accès aux données, sont prises en charge par un thread dédié qui assure la synchronisation avec l'application locale et la communication avec les applications distantes (Fig. 1).

EPSN utilise des signaux et des sémaphores qui permettent de synchroniser l'exécution du processus avec le thread dédié et de garantir un accès cohérent aux données, sans copie via la mémoire partagée. Les communications distantes reposent sur un mécanisme client/serveur CORBA entièrement masqué à l'utilisateur. Les objets CORBA hébergés par le serveur implantent les diverses fonctionnalités de la plate-forme et les références CORBA permettent de manipuler ces objets via un mécanisme d'invocation de procédures distantes reposant sur le bus logiciel de CORBA (Object Request Broquer). On attribue souvent à CORBA de faibles performances, pourtant il existe des implantations pratiquement aussi efficaces que MPI sur réseau rapide comme le montre [10]. Le noyau (*core*) est l'élément central de la plate-forme, qui coordonne les différents traitements – il a une visibilité complète sur l'application locale, les objets et les références CORBA.

Dans plusieurs environnements de steering, comme CUMULVS, une phase de communication, bloquant la simulation, est explicitement appelée dans l'instrumentation du code. Dans *EPSN*, au contraire, les communications sont commandées par le thread qui se base sur l'instrumentation pour savoir si les données sont accessibles. Afin de garantir un accès asynchrone aux données distantes, *EPSN* utilise des requêtes CORBA *oneway* et un mécanisme d'objets *callback* afin de recevoir les données. Cette architecture asynchrone associée aux accès en mémoire partagée permettent le recouvrement des communications.

3. Le prototype *Epsilon*

Le prototype *Epsilon* se compose de deux bibliothèques de fonctions C/Fortran : l'API simulation qui permet d'instrumenter les simulations numériques conformément au modèle abstrait et l'API cliente qui permet la construction de client de pilotage et de visualisation. Cette dernière API est également disponible dans les langages Perl, Python et Java. *Epsilon* est écrit en langage C++ et se base sur *omniORB4* [11], une implantation de CORBA offrant de bonnes performances.



(a) `get/wait` (b) `getp/cancelp`

Figure 3: Schémas de communication & synchronisation pour quelques fonctions *Epsilon*.

3.1. Fonctionnalités et schémas de communication

Contrôle – Le contrôle d'une simulation repose sur le positionnement de fonctions `barrier` dans son code source. Ces fonctions se comportent comme autant de points d'arrêt dont l'état, "bloquant" ou "passant", est programmable à distance par le client (appel à la fonction `setbarrier`). En outre le client dispose des commandes classiques `play/step/stop` pour contrôler de manière globale le déroulement de la simulation. L'évolution temporelle de la simulation est matérialisée par des compteurs de boucle incrémentés par la fonction `iterate`, qui joue également le rôle d'une barrière implicite pour le contrôle de la boucle.

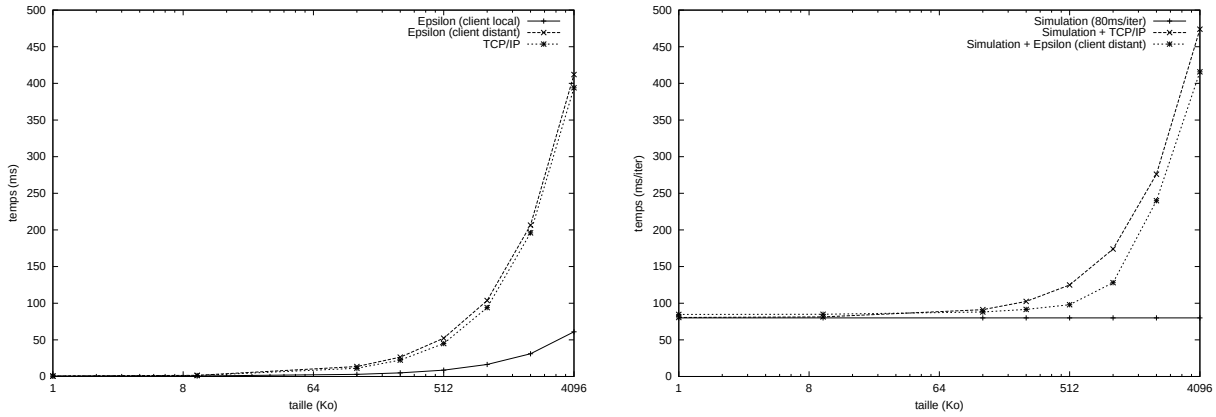
Extraction des données – L'application cliente peut extraire les données de la simulation grâce à la fonction `get` (Fig. 3(a)). L'appel à cette fonction est traduit par une requête CORBA, gérée par le client grâce aux fonctions `wait` et `test`. Du côté simulation, l'accès aux données est protégé par les fonctions `lock/unlock` qui délimitent des plages d'accessibilité dans le code source de la simulation. Par conséquent, l'envoi d'une donnée peut suivre immédiatement la réception de la requête CORBA `get` ou être différé si la donnée n'est pas encore accessible (Fig. 3(a)). Une fois la donnée reçue par le client, elle peut être automatiquement copiée dans l'espace mémoire du client ou bien déclencher un traitement prédéfini selon un mécanisme de *callback*. Notons que ces traitements sont identiquement protégés par des plages d'accès `lock/unlock` dans le code client.

Le client peut aussi requérir une donnée de manière *permanente* avec la fonction `getp` afin de recevoir continûment les nouvelles versions de cette donnée (Fig 3(b)) et de produire de la visualisation "en temps réel". Les nouvelles versions des données sont signalées explicitement par la fonction `release` ou implicitement par la fonction `iterate`. Un système d'acquittement (requête CORBA `ackp`) régule automatiquement le transfert de données, en sautant volontairement l'envoi de certaines versions pour éviter un engorgement du client. Par ailleurs, une fonction `flush` permet de forcer l'envoi systématique de la donnée à chaque nouvelle version, en ralentissant volontairement la simulation si nécessaire.

Modification des données – De manière symétrique, le client peut modifier à distance une donnée de la simulation en invoquant la fonction `put` qui transfère simplement sa version locale de la donnée vers la simulation, tout en respectant les plages d'accessibilité.

3.2. Extension au parallélisme

La représentation abstraite des simulations, telle qu'elle est présentée dans la section 2.1, s'étend aux simulations parallèles (SPMD). Dans ce modèle, une simulation est considérée comme arrêtée dans un état stable si chaque processus est arrêté sur le même point d'arrêts de la même itération. De même, une donnée de la simulation est considérée comme cohérente si chaque élément distribué qui la compose se trouve dans un état cohérent et s'ils ont la même version. Pour traiter les simulations SPMD,



(a) Temps de communication.

(b) Extraction de données d'une simulation.

Figure 4: Evaluation des performances du prototype *Epsilon*.

l'infrastructure *EPSN* contrôle et coordonne un ensemble de processus en garantissant ces propriétés.
Nous avons étendu le prototype séquentiel aux simulations SPMD, en utilisant une infrastructure de communication basée uniquement sur CORBA. Chaque processus de la simulation est instrumenté avec un thread *Epsilon* indépendant. De plus, nous avons introduit un objet CORBA spécifique sur le processus 0 servant de point d'entrée aux clients et permettant d'obtenir une vision globale de la simulation parallèle. Pour garantir la cohérence des opérations d'*Epsilon*, nous avons développé des protocoles de synchonisation : les requêtes du client sont transmises à tous les processus concernés, chacun s'arrête indépendamment sur le premier point d'arrêt rencontré puis se synchronisent sur la position la plus avancée de la simulation, avant de traiter effectivement la requête.
L'utilisation de PaCO++ [12, 13], une extension parallèle des objets CORBA, devrait améliorer ces phases de synchronisation grâce à l'utilisation d'une couche de communication interne performante. Sur la figure 2, nous voyons que cette architecture présente trois niveaux de communication : les processus parallèles communiquent entre eux, les objets parallèles CORBA communiquent à travers une couche de communication interne (généralement en MPI) et enfin, les applications clientes communiquent avec la simulation via l'ORB. Par ailleurs, les objets parallèles CORBA proposent une gestion transparente de la redistribution des données vers des clients pouvant eux-mêmes être parallèles.

3.3. Evaluation des performances

Nous présentons dans cette section une évaluation des performances du prototype *Epsilon*. Ces résultats proviennent d'expériences réalisées sur deux PCs (Pentium IV) reliés par un réseau Ethernet 100 Mb/s.
La figure 4(a) donne le temps de communication moyen d'*Epsilon* pour différentes tailles de données (de 1Ko à 4Mo) dans le cas d'un client local ou distant (requête `getp`). On remarquera que le surcoût en communication d'*Epsilon* n'est pas très élevé, comparé à TCP/IP au dessus duquel *omniORB4* communique.
L'expérience de la figure 4(b) présente le cas test d'une simulation effectuant 80 ms de calculs par itération avec une donnée accessible la moitié du temps et d'un client distant recevant cette donnée à chaque pas de temps (requête `getp`). La figure montre le temps moyen d'une itération de la simulation pour différentes tailles de données. Pour mettre en évidence les possibilités de recouvrement du prototype, cette courbe est comparée à la somme théorique du temps de calcul d'une itération et du temps d'envoi de la donnée par TCP/IP. On notera que pour des données de taille inférieure à 64 Ko, le surcôut mesuré est faible alors que pour des tailles supérieures, le surcoût est largement compensé par le recouvrement des communications. Par ailleurs, cette expérience montre un surcoût d'instrumentation hors communication (avec ou sans client) de l'ordre de la milliseconde.

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

<simulation name="fingerprint" context="sequential">
  <nameservice host="kamille" port="2809"/>

  <control running="true">
    <breakpoint id="fingerprint_init"/>
    <loop id="deflation" state="down">
      <breakpoint id="deflation_mid" state="up"/>
    </loop>
    <breakpoint id="deflation_end" state="down"/>
    <breakpoint id="fingerprint_end" state="down"/>
  </control>

  <data>
  <group id="mesh" >
    <scalar id="nb_nodes" access="readonly" type="long"/>
    <scalar id="nb_cells" access="readonly" type="long"/>
    <sequence id="nodes" access="readwrite" type="double">
      <dimension size="3"/>
      <dimension size="nb_nodes"/>
    </sequence>
    <sequence  id="cells" access="readwrite" type="long">
      <dimension size="3"/>
      <dimension size="nb_cells"/>
    </sequence>
    <sequence id="values" access="readwrite" type="double">
      <dimension size="nb_nodes"/>
    </sequence>
  </group>
  </data>

</simulation>
```

Figure 5: Description XML de *FingerPrint*.

```
// Initialisation de la molécule et du maillage initial
int nb_nodes, nb_cells; int ** nodes, ** cells; double * values;
initMoleculeAndBuildMesh("crambin.pdb");

// Initialisation d'Epsilon
epsilon_init("fingerprint.xml");
epsilon_publish("nb_nodes", &nb_nodes);
epsilon_publish("nb_cells", &nb_cells);
epsilon_publish("nodes", nodes);
epsilon_publish("cells", cells);
epsilon_publish("values", values);
epsilon_publishgroup("mesh");
epsilon_unlockall();
epsilon_barrier("fingerprint_init");

// Tant que le maillage ne colle pas à la surface
while (deflatTest()) {
  epsilon_iterate("deflation");
  epsilon_lock("nodes");
   moveNodes(); // déplacement des noeuds vers la surface
  epsilon_unlock("nodes");
  evaluateMesh(); // evaluation du maillage
  epsilon_barrier("deflation_mid");
  epsilon_lock("mesh");
  correctMesh(); // correction du maillage
  epsilon_unlock("mesh");
  epsilon_flush("mesh");
}
epsilon_barrier("deflation_end");

// Construction de l'empreinte à partir du maillage
makePrint();
epsilon_barrier("fingerprint_end");
epsilon_exit();
```

Figure 6: Instrumentation de *FingerPrint*.

4. Développement d'une simulation interactive

Cette section présente les différentes étapes nécessaires au développement d'une simulation interactive avec *EPSN*, en partant du code de simulation jusqu'à la visualisation et l'interaction. Notre propos est illustré avec l'exemple d'une simulation en chimie moléculaire, *FingerPrint* [14]. Cette simulation est écrite en C et s'inscrit dans un projet plus général sur le criblage de molécules pour du docking moléculaire. Ce code calcule d'abord un maillage de la surface d'un système moléculaire, puis son empreinte. Nous nous intéressons ici plus particulièrement au pilotage de la construction du maillage par une méthode de déflation.

4.1. Description XML de la simulation

L'instrumentation d'une simulation commence par la création d'un fichier XML la décrivant. Ce fichier est interprété à l'initialisation de la simulation (appel à la fonction `init`) et sa description est automatiquement transmise à l'initialisation du client. Ainsi cette description est la seule représentation de la simulation partagée par la simulation et tous ses clients.

Comme le montre l'exemple de *FingerPrint* (Fig. 5), le fichier XML contient à la fois la description des éléments de contrôle (section `control`) et des données exportées (section `data`). La première section regroupe les informations relatives aux boucles de calcul (`loop`) et aux points d'arrêt (`breakpoint`). Les données numériques décrites dans la deuxième section peuvent être des constantes (`parameter`), des scalaires (`scalar`) ou des séquences (`sequence`). A chaque donnée est associée un identifiant, un type, les droits d'accès et la valeur initiale pour les constantes. Pour les séquences qui s'apparentent à des tableaux, il faut encore détailler chacune des dimensions (`dimension`) en précisant la taille, les offsets à gauche et à droite. La description XML permet également de regrouper logiquement un ensemble de données (`group`) pour définir des objets plus complexes comme par exemple le maillage *mesh* enveloppant la molécule. En outre, on trouve des informations utiles pour le déploiement des clients, avec notamment le nom de la simulation et la localisation du service de nommage CORBA (`nameservice`).

4.2. Instrumentation

Comme nous l'avons déjà dit, l'intégration d'une simulation dans *EPSN* est réalisé par des appels de fonctions de la bibliothèque *Epsilon*, reprenant en argument les éléments décrits dans le XML (*loop*, *breakpoint*, *data*). La figure 6 présente l'instrumentation du code simplifié de *FingerPrint*. Ce code se découpe classiquement en trois phases : l'initialisation, le corps de la simulation avec la boucle de calcul

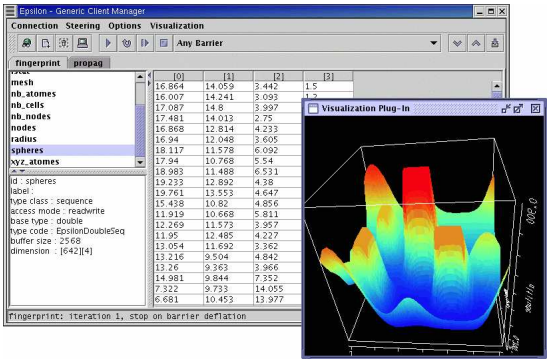


Figure 7: Client générique d'*EPSN*.

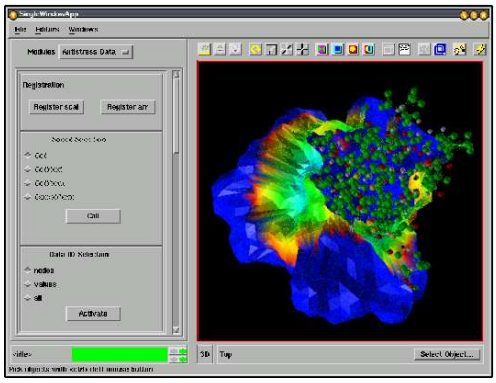


Figure 8: Client semi-générique (*FingerPrint*).

principale et la terminaison.

L'initialisation de toute l'infrastructure *EPSN* se fait simplement à l'appel de la fonction `init` qui prend en argument le fichier XML décrivant la simulation (Fig. 5). C'est à ce moment que la simulation s'inscrit auprès du service de nommage CORBA sous le nom *fingerprint* et devient visible sur le réseaux pour un client de pilotage *EPSN*. L'utilisateur publie les données décrites dans le XML pour les rendre accessibles aux clients. Pour ce faire, il pointe chacune d'elles dans la mémoire du processus (`publish`).

Dans le corps de la simulation, l'utilisateur marque l'évolution de la boucle de calcul *deflation* avec la fonction `iterate` et place les divers points d'arrêt (`barrier`). Les zones d'accès aux données sont matérialisés par les fonctions `lock/unlock`. A la fin de l'initialisation, la simulation s'arrête automatiquement sur le point d'arrêt *fingerprint_init* où toutes les données sont consultables (`unlockall`). Lors de la boucle de calcul, on voit par exemple que l'accès aux noeuds (*nodes*) est protégé pendant l'appel à la fonction `moveNodes`, qui modifie effectivement cette donnée, alors que les autres données restent consultables.

Enfin la fonction `exit` de la bibliothèque *Epsilon* libère toutes les ressources et termine proprement le thread *Epsilon*. Après quoi, la simulation continue librement son exécution et toute requête cliente lève une exception CORBA, qui s'achève par un retour d'erreur.

4.3. Client de pilotage & visualisation

Une fois l'instrumentation réalisée, l'utilisateur se sert d'un client de pilotage pour interagir avec la simulation à distance. Les clients de pilotage intègrent classiquement des fonctionnalités de visualisation pour suivre en temps réel l'évolution des données et les manipuler. *EPSN* propose trois stratégies complémentaires pour piloter une simulation instrumentée.

Une première approche consiste à développer un client spécifique, adaptée à une simulation donnée. Il est possible de travailler directement à partir de l'interface CORBA (IDL), mais il est plus pratique d'utiliser les fonctionnalités de l'API cliente (cf. 3.1). Nous avons validé cette approche en la combinant à des bibliothèques tels que OpenGL, OpenInventor ou VTK pour produire de la visualisation en "temps réel". Une deuxième approche consiste à utiliser le client générique d'*EPSN*, entièrement développé en Java. Cet outil possède un interface graphique qui permet de contrôler et d'accèder aux données de plusieurs simulations simultanément. Les données sont présentées au travers de simples tables numériques ou des "plug-ins" de visualisation (Fig. 7). Une approche intermédiaire consiste à utiliser des clients semi-génériques, dédiés à la visualisation interactive des objets complexes (e.g. les maillages non structurés, les molécules, etc.). Cette approche repose sur l'utilisation d'un ensemble de modules génériques permettant l'acquisition et la configuration de données représentant ces objets. En outre, elle permet d'exploiter des outils de visualisation avancée comme *AVS/Express* [15] (Fig. 8).

5. Conclusion

La plate-forme *EPSN* fournit une approche générique, flexible et dynamique pour le pilotage des simulations. Elle repose sur une instrumentation du code peu intrusive, qui ne remet pas en cause son organisation. L'exemple de *FingerPrint* a démontré que quelques dizaines de lignes suffisent à rendre la simulation interactive. De plus, les bonnes performances du prototype séquentiel *Epsilon* valident le choix de la technologie CORBA et d'une architecture mettant en oeuvre un recouvrement des communications. Il reste encore à démontrer la scalabilité de cette approche pour le pilotage des codes parallèles. Dans cette optique, les récents développements autour des objets parallèles CORBA nous offre de bonnes perspectives d'évolution.

Les développements en cours d'*EPSN* sont orientés vers la gestion d'applications interactives complètement parallèles, tant dans la partie simulation que dans la partie visualisation, et vers les problèmes sous-jacents liés à la redistribution des données régulières et complexes (maillage, molécule, etc.). Cette évolution constitue une étape décisive pour le pilotage des simulations sur la grille.

Bibliographie

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. U. Lang, J.P. Peltier, P. Christ, S. Rill, D. Rantzau, H. Nebel, A. Wierse, R. Lang, S. Causse, F. Juaneda, M. Grave, and P. Haas. *COVISE: Perspectives of Collaborative Supercomputing and Networking in European Aerospace Research and Industry*, volume 11. Future Generation Computer Systems (FGCS) Elsevier Science, 1995.
4. 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.
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. J. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proceedings of the 11th IPPS'97*, pages 128–132, 1997.
7. 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.
8. Greg Eisenhauer, Beth Plale, and Karsten Schwan. DataExchange: high performance communications in distributed laboratories. *Parallel Computing*, 24(12–13):1713–1733, 1998.
9. 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.
10. 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.
11. omniORB home page. http://omniorb.sourceforge.net.
12. 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.
13. Alexandre Denis, Christian Pérez, and Thierry Priol. Achieving portable and efficient parallel CORBA objects. *Concurrency and Computation: Practice and Experience*, 2003. To appear.
14. Cai W.; Shao X.; Maigret B. Protein-ligand recognition using spherical harmonic molecular surfaces: Towards a fast and efficient filter for large virtual throughput screening. *J. Molec. Graph. Model.*, 20:313–328, 2002.
15. Advanced Visual Systems Inc. Avs/Express visualization edition. http://www.avs.com, 2003.