Conception et mise en oeuvre d'une plate-forme de pilotage de simulations numériques parallèles et distribuées

Nicolas Richart

▶ To cite this version:

Nicolas Richart. Conception et mise en oeuvre d'une plate-forme de pilotage de simulations numériques parallèles et distribuées. Informatique [cs]. Université Sciences et Technologies - Bordeaux I, 2010. Français. NNT : . tel-00464406

HAL Id: tel-00464406

https://theses.hal.science/tel-00464406v1

Submitted on 17 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre : 3991

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Nicolas RICHART**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Conception et mise en œuvre d'une plate-forme de pilotage de simulations numériques parallèles et distribuées.

Soutenue le : 20 janvier 2010

Après avis des rapporteurs :

Michel Kern	Chargé de recherche
cosigné par Jérôme Jaffré	Directeur de recherche
Christian Perez	Chargé de recherche

Devant la commission d'examen composée de :

Olivier Coulaud	Directeur de Recherche	Directeur de Thèse
Michel Kern	Chargé de recherche ...	Examinateur
Raymond Namyst	Professeur	Président et rapporteur du jury
Christian Perez	Chargé de recherche ...	Examinateur
Jean Roman	Professeur	Directeur de Thèse
Sophie Valcke	Ingénieur CERFACS ..	Examinatrice

– 2010 –

Remerciements

Je souhaite tout d'abord remercier les membres de mon jury, en commençant par ceux qui ont acceptés d'être rapporteur, Michel Kern et Christian Perez. Je remercie également Jérôme Jaffré qui a accepté de co-signer le rapport de Michel Kern. Merci à Sophie Valcke et Raymond Namyst d'avoir accepté de faire partie de mon jury de thèse.

Enfin merci à Olivier Coulaud et Jean Roman de m'avoir proposé de faire cette thèse et de m'avoir encadré pendant ces nombreuses années. Merci à Olivier pour sa patience et ses conseils durant ces années, et pour avoir porté EPSN sur de nombreuses architectures ésotériques dont personne dans l'équipe ne voulait entendre parler (ex. Windows). Merci à Jean d'avoir sacrifié une pleine brouette de stylos rouges, si craints auprès de thésards passés et futurs.

Je voudrais remercier tout particulièrement Aurélien Esnard qui a participé à l'encadrement de ma thèse. Il a été en quelque sort mon maître Jedi (ou mon seigneur Sith je sais pas trop), sous le nom de DarkOrel. Il m'a transmis son « *bébé* », EPSN pour que je me l'approprie et que je le fasse évoluer dans le cadre de mes travaux.

Je voudrais également remercier tous ceux qui ont eu le courage de relire ma thèse pour essayer d'épargner les stylos rouges de Jean, Guilhem et Damien.

Pour continuer mes remerciements, je tiens a dire merci à tous mes co-bureaux et qui ont eu à me supporter. Ils furent nombreux avec tous les déménagements. Il y a tout d'abord eu le bureau très « *chaleureux* » du Haut-Carré avec Orel et Mick où j'ai effectué mon PFE durant lequel Olivier m'a proposé de continuer en thèse. Puis il y a eu le déménagement au LaBRI au sein de la team « *super-serieux* » Gaple, Cheche, Benji puis dans le bureau en tête-à-tête avec Guillaume et une horloge Gentoo, puis Mathieu. Ensuite il y a eu le déménagement dans les premiers modulaires, dans le bureau avec Adam et Robin, où l'on ne se comprenait pas toujours (ils me parlaient courbes de convergence et résidu, je leur répondais CORBA :)). Et enfin il y a eu mon dernier bureau avec Peter.

Mais mon expérience dans l'équipe ScAlApplix et HiePACS ne se résume pas qu'à mes co-bureaux, il y a eu aussi tout ceux que j'ai côtoyé et avec qui on a passé de bons moments, Christophe (le mexicain cinéphile), la spong team : Pierre Pascal et Olivier (dit le boucher), François, Abdou, Mario, Babeth, Paulette et Juliette, Mathieu (le cowboy), Jérémie, Aurélien et Christophe (ma WoW team), XL, Cacal, Gonnet, Cécile et Raton, Christelle, Jirka, Algiane (ou Rachel), $\cdots$

Parmi ces bons moments il y a eu les parties « *anti-stresse* » de Tremulous, de DotA et bien sur de Liquidwar. Mais il y a aussi eu les petites soirées à « *la cantine* » (le Palatium) suivies quelques fois du Chabrot ou d'un petit Gabi.

Merci également à tout ceux qui m'ont aidé à réaliser mes travaux dans les meilleurs conditions possibles, essentiellement en déménageant un peu partout le cluster graphique et en supportant mes demandes sur Grid5000, Mathieu (dit masouche), Peyo, JP, Jérôme B et C, Florent, Julien.

Et pour finir un grand merci à maman de m'avoir soutenu toutes ces années. Un grand merci aussi d'avoir fait le déplacement avec Ray jusqu'à Bordeaux pour ma soutenance et d'avoir préparé un super pot :). Merci à ma soeur et à Nicolas d'avoir fait de moi le parrain du petit Edgar (ouais ça a rien à voir directement avec ma thèse mais c'est bien du bonheur).

Et comme dirait un philosophe contemporain, « *Eh ! Mec ! La thèse c'est de l'amour Mec ! – Hubert L.* »

Résumé

Le domaine de la simulation numérique évolue vers des simulations de phénomènes physiques toujours plus complexes. Cela se traduit typiquement par le couplage de plusieurs codes de simulation, où chaque code va gérer une physique (simulations multi-physiques) ou une échelle particulière (simulations multi-échelles). Dans ce cadre, l'analyse des résultats des simulations est un point clé, que ce soit en phase de développement pour valider les codes ou détecter des erreurs, ou en phase de production pour confronter les résultats à la réalité expérimentale. Dans tous les cas, le pilotage de simulations peut aider durant ce processus d'analyse des résultats. L'objectif de cette thèse est de concevoir et de réaliser une plate-forme logicielle permettant de piloter de telles simulations. Plus précisément, il s'agit à partir d'un client de pilotage distant d'accéder ou de modifier les données de la simulation de manière cohérente, afin par exemple de visualiser "en-ligne" les résultats intermédiaires. Pour ce faire, nous avons proposé un modèle de pilotage permettant de représenter des simulations couplées et d'interagir avec elles efficacement et de manière cohérente. Ces travaux ont été validés sur une simulation multi-échelles en physique des matériaux.

Mots-clés: pilotage de simulation, simulation numérique, parallélisme, couplage de codes, modélisation de simulations.

Abstract

The numerical simulations evolve more and more to simulations of complex physical phenomena through multi-scale or multi-physics codes. For these kind of simulations data analysis is a main issue for many reasons, as detecting bugs during the development phase or to understand the dynamic of the physical phenomena simulated during the production phase. The computational steering is a technique well suited to do all this kind of data analysis. The goal of this thesis is to design and develop a computational steering framework that take into account the complexity of coupled simulations. So, through a computational steering client we want to interact coherently with data generated in coupled simulations. This afford for example to visualize on-line the intermediate results of simulations. In order to make this possible we will introduce an abstract model that enables to represent coupled simulations and to know when we can interact coherently with them. These works have been validated on a legacy multi-scale simulation of material physics.

Keywords: Computational steering, numerical simulation, parallelism, code coupling, simulation modelisation.

Table des matières

Introduction générale **1**

1 État de l'art et positionnement **7**

1.1 Simulations numériques parallèles distribuées . . . 8
1.1.1 La simulation numérique . . . 8
1.1.2 Simulations numériques distribuées . . . 9
1.2 Visualisation scientifique . . . 11
1.2.1 Principes de base . . . 11
1.2.2 Techniques de visualisation parallèle . . . 13
1.3 Les bases des environnements de couplage . . . 16
1.3.1 La couche de communication . . . 17
1.3.2 La couche de redistribution . . . 19
1.3.3 Les modèles de haut niveau . . . 22
1.4 État de l'art sur le pilotage de simulations . . . 27
1.4.1 Introduction au pilotage de simulations . . . 27
1.4.2 Environnements de pilotage existants . . . 28
1.4.3 Les limites des environnements existants . . . 36
1.5 Positionnement . . . 38
1.5.1 Modélisation des simulations couplées . . . 39
1.5.2 Cohérence des traitements . . . 39

2 Modèle pour un environnement de pilotage **41**

2.1 Introduction . . . 42
2.2 Modèle abstrait de EPSN . . . 42
2.2.1 Modèle de description de simulations SPMD . . . 42
2.2.2 Modèle de pilotage . . . 48
2.2.3 Limitations du modèle pour les simulations distribuées . . . 52
2.3 Modèle abstrait pour les simulations distribuées . . . 53
2.3.1 Modèle de description de simulations distribuées . . . 53

2.3.2 Modèle de pilotage 62
2.4 Conclusion 70

3 Réalisation **71**
3.1 Introduction 72
3.2 La plate-forme EPSN2 72
3.2.1 Vue d'ensemble 73
3.2.2 Intégration d'une simulation distribuée dans EPSN 77
3.2.3 Implantation des différents algorithmes et schémas de communication . . 79
3.3 Client de pilotage et de visualisation 86
3.3.1 Clients EPSN 86
3.4 Conclusion 91

4 Validation **93**
4.1 Introduction 94
4.2 Évaluation de la bibliothèque ColCOWS 96
4.2.1 Activation d'un workspace 96
4.2.2 Communications collectives 97
4.3 Test de validation pour une simulation M-SPMD 99
4.3.1 Initialisation de la plate-forme 99
4.3.2 Algorithmes de coordination 100
4.3.3 Client de visualisation parallèle 103
4.4 Test de validation pour des simulations distribuées réelles 104
4.4.1 LibMultiScale : simulation couplée de type M-SPMD 105
4.4.2 Couplage DLPoly/Siesta 109
4.5 Conclusion 111

Conclusion et perspectives **113**

Annexes **117**

A API de EPSN **117**
A.1 Descritption XML complète de la simulation Δ 117
A.2 Différents listings lié a la simulation LibMultiScale 119

B Bibliographie principale **123**

C Liste des publications **133**

Glossaire **135**

Introduction générale

L'augmentation de la puissance de calcul mise à la disposition des scientifiques au travers des calculateurs hautes performances et des grilles de calcul ouvre d'importantes perspectives de recherche dans le domaine de la simulation numérique de phénomènes complexes. Ainsi ces simulations numériques sont de plus en plus fines au niveau des modèles et de plus en plus précises en combinant ces modèles physiques entre eux. On parle de simulations multi-physiques et/ou multi-échelles. Un couplage de modèles physiques et donc mathématiques, se traduit assez naturellement par un couplage de codes de simulation. Les simulations modernes en vraie grandeur ne sont donc plus de « *simples* » codes parallèles simulant un phénomène physique, mais elles évoluent vers des compositions de plusieurs codes parallèles où chaque code simule un modèle physique particulier. Ce type de simulation permet alors de réutiliser les différents codes parallèles existants. Comme exemple d'applications couplées, on présente souvent des simulations de l'évolution climatique de la Terre comme celles des simulations du projet CCSM [21] (Community Climate System Model). Cette simulation est un couplage de quatre codes ayant pour but de simuler plus finement les phénomènes climatiques à grande échelle ; elle comprend un code gérant l'atmosphère (CAM [20] – Community Atmosphere Model), un pour les océans (POP [117] – Parallel Ocean Program), un pour la glace (CSIM [28] – Community Sea Ice Model) et le dernier pour prendre en compte la modélisation de la terre et de la végétation (CLM [23] – Community Land Model) ; ce couplage de modèle est présenté sur la figure 1. Historiquement, la simulation était composée uniquement d'un couplage atmosphère/océan et afin de raffiner la modélisation du système climatique, un modèle terre a été ajouté à la simulation. Toujours dans le but de raffiner la modélisation le code modélisant la glace a été intégré.

Les simulations deviennent donc de plus en plus complexes et demandent de plus en plus de ressources, que ce soit en termes de puissance de calcul ou de taille mémoire. Cela implique aussi que l'analyse des résultats de ces simulations se complexifie. Les procédés utilisés pour traiter les résultats doivent donc prendre en compte les importants volumes de données pouvant être générés, ainsi que la complexité en mémoire de ces dernières. En effet, les données peuvent provenir de plusieurs codes de simulation à la fois, mais elles peuvent également avoir une représentation différente d'un code à l'autre. En général, de telles simulations sont exécutées en mode « *batch* » et les données sont enregistrées dans des fichiers pour être traitées ultérieurement. Afin d'avoir un retour direct et aussi interactif que possible sur la simulation, nous pouvons avoir recours à des techniques de pilotage de simulations. Dans le cadre du pilotage de simulations et de la visualisation « *en ligne* » des données, il faut prendre en compte les difficultés dûes à la nature des données, afin d'assurer la cohérence de celles-ci, et des interactions avec les simulations. La visualisation parallèle est un moyen efficace de traiter ces volumes importants de données. Cette solution est d'autant plus efficace et aisée à mettre en place que les grappes d'ordinateurs se démocratisent. Il faut tout de même noter que ce type de visualisation rend le problème de cohérence des données dans le procédé de traitement des données encore plus complexe.

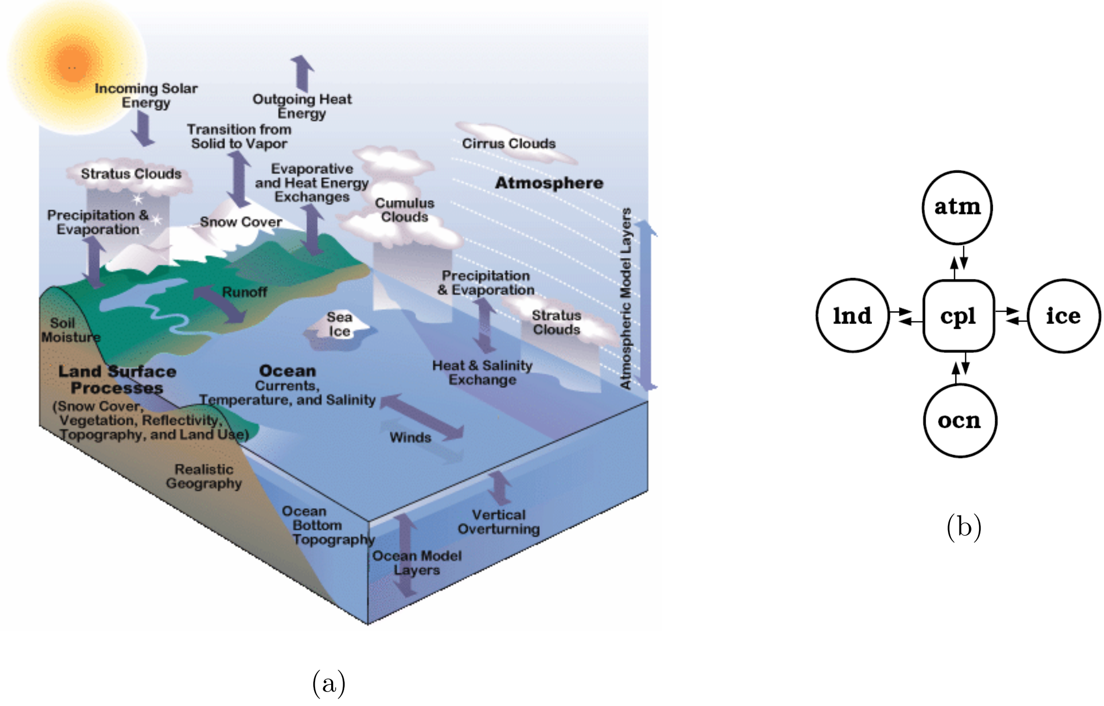


FIGURE 1 – Couplage de modèles (a) et de codes (b) dans CCSM

Différents types de couplage

Dans le cadre des simulations HPC (High Performance Computing), les couplages de simulations se rencontrent essentiellement dans certains domaines applicatifs, comme la météorologie ou la physique des matériaux. Ceci peut s'expliquer par le haut niveau de complexité des modèles physiques de ces domaines. Par la suite, nous allons détailler différents types de couplages. Nous en distinguerons essentiellement deux : les couplages multi-physiques et les couplages multi-échelles.

Le couplage multi-physique se rencontre souvent dans les simulations de climatologie ou de météorologie comme celles que nous avons présentées au début de ce chapitre. Mais, il ne se limite pas à ces domaines applicatifs et on trouve également de nombreux couplages fluide/structure. Dans ce dernier cas, il s'agit d'étudier l'influence de l'écoulement d'un fluide sur une structure mécanique et les déformations que cela peut engendrer, comme par exemple l'écoulement de l'air autour d'un profil d'aile d'avion, l'écoulement du sang dans les artères. La figure 2 illustre un exemple de couplage thermique/éléctromagnétisme/structure pour la simulation d'un moteur électrique faite avec le code COMSOL [24].

Dans ce type de simulation, il s'agit donc de coupler plusieurs codes gérant chacun une physique différente. Ces physiques sont souvent calculées sur des domaines différents, ayant des interfaces communes, mais elles peuvent également être définies sur un même domaine. Par exemple, dans le cas de couplage océan/atmosphère, il s'agit d'un couplage multi-physiques, multi-domaines et l'interface commune se situe à la surface de l'eau; mais si l'on considère le couplage océan/biogéochimie, le domaine physique est le même et c'est donc un couplage de type multi-physiques, mono-domaine. Dans tous les cas, la nature des données dans les codes est différente, soit du fait des domaines d'application qui diffèrent, soit simplement à cause des

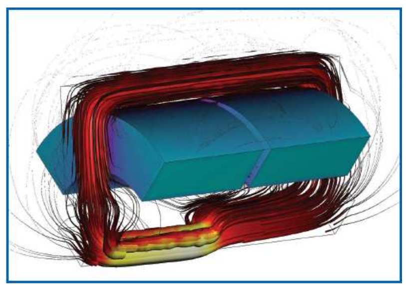


FIGURE 2 – Exemple de simulations multi-physiques : couplage thermique/éléctro-magnétisme/structure dans un moteur électrique

variables nécessaires aux calculs. Lors du post-traitement des résultats, il faut donc pouvoir gérer des données de types totalement différents, mais représentant potentiellement une même information. Par conséquent, il faut pouvoir assurer la cohérence entre ces différentes données.

Les couplages multi-échelles, quant à eux, sont basés sur des couplages de codes gérant le même phénomène physique mais à des échelles différentes. Ce type de simulation est très présent dans la physique des matériaux notamment pour capturer et comprendre certains phénomènes physiques fins. Il faudrait considérer une échelle très petite ; or plus l'échelle est petite et plus les calculs sont coûteux et moins le domaine d'étude est grand. Pour remédier à ce problème, on couple différentes échelles afin de pouvoir obtenir la finesse nécessaire pour simuler le phénomène physique sans que cela soit trop coûteux pour autant. Une autre raison est que dans certains cas, le phénomène que l'on souhaite étudier nécessite une étude à différentes échelles pour pouvoir être pleinement modélisé. Dans le cas de l'étude de la propagation des fissures [5], le phénomène va être simulé à l'échelle atomique (microscopique) autour de la zone de fissure (de discontinuité). Mais pour que les calculs ne soient pas trop coûteux loin de la discontinuité, le matériau va être simulé à une échelle macroscopique. On trouve également ce type de couplage de simulation dans le domaine de la physique des matériaux et de la biologie, avec des méthodes de calcul QM/MM (Quantum Mechanics/Molecular Mechanics). Ces méthodes couplent des méthodes de mécanique quantique à de la mécanique moléculaire et ce afin d'avoir une modélisation plus fine dans le domaine quantique. Tout comme pour les couplages multi-physiques, les données vont être de nature parfois très différentes dans les divers codes utilisés. Même si la physique modélisée est la même, les domaines et les grandeurs physiques ne sont pas les mêmes. Une même information pourra ne pas être représentée de la même façon dans les différents codes. Par exemple dans la simulation étudiant la propagation des fissures, on voudra visualiser le déplacement de la matière ; cela se traduit par la différence entre la position initiale des atomes et leurs positions actuelles dans la zone de la fissure et par une variable de déplacement associée aux nœuds du maillage dans la zone plus éloignée.

Dans ces couplages multi-physiques et/ou multi-échelles, les motivations sont donc assez proches : modéliser plus finement les phénomènes physiques tout en limitant les coûts de calcul. Nous avons également vu que dans ces deux cas, les données différaient entre les codes. Les difficultés pour représenter ces données comme une seule et même information seront donc les mêmes lors des post-traitements ou lors de la visualisation « *en ligne* » dans le cadre du pilotage de simulation.

Pilotage cohérent de simulations numériques complexes

Habituellement, les simulations qu'elles soient parallèles ou distribuées sont lancées en mode « *batch* », c'est-à-dire que l'exécution des simulations est planifiée à l'avance sur des calculateurs. Quand leurs exécutions débutent, elles produisent des fichiers de résultats et une fois l'exécution terminée, l'utilisateur peut récupérer ces fichiers afin de les analyser. Dans ce cas, on parlera de post-traitement (ou *post-processing*). Dans cette approche, les simulations génèrent de gros volumes de données qu'il faut déplacer sur des machines adaptées autant que faire ce peut au post-traitement des résultats. et cette étape peut s'avérer très fastidieuse. Une fois les données analysées, et si un paramètre de la simulation s'avère ne pas être bon, il faudra tout recommencer depuis le début : planification de l'exécution, exécution, récupération des données et analyse.

Le pilotage de simulation (ou *computation steering*) a pour but de faciliter le processus de simulation de l'exécution à l'analyse. Cette méthode permet de ne pas attendre la fin de l'exécution d'une simulation pour analyser les résultats et ce en visualisant « *en ligne* » et à distance les données produites par la simulation. Si l'utilisateur se rend compte que sa simulation ne va pas dans la « *bonne direction* » (analyse de la dynamique de la simulation), il peut modifier « *à la volée* » les paramètres de la simulation, sans avoir à la relancer. Ainsi cette méthode permet d'augmenter la « *productivité* » et l'efficacité de l'analyse des résultats en réduisant le temps entre les changements de paramètres (analyse de sensibilité) et la visualisation des modifications. Mieux encore, elle permet de retrouver une démarche expérimentale par l'interaction qu'elle offre avec la simulation. Le pilotage de simulation ne s'oppose pas aux techniques de post-traitements pour autant. Bien au contraire ces deux techniques peuvent se compléter.

Néanmoins, le pilotage de simulations est plus complexe à mettre en œuvre que le simple post-traitement des fichiers de résultats. Dans le cas de simulations parallèles, les données sont réparties sur plusieurs processus. Dans le cas de simulations distribuées, elles sont également réparties entre les différents codes et n'ont pas la même nature dans tous les codes. Dans ce cas, il faut pouvoir assurer la cohérence des données à traiter. Par exemple, dans le cas de la visualisation des données, on souhaitera afficher la même information quelle que soit sa répartition et sa nature dans les codes de simulation. Cela peut être fait de façon « *naïve* », par exemple en synchronisant fortement les différents processus, mais cela peut s'avérer très coûteux dans le cadre de simulations HPC. D'autres méthodes comme la planification des traitements permettent d'assurer la cohérence des traitements en étant moins intrusif pour les simulations, mais ces méthodes sont plus difficiles à mettre en œuvre.

Objectifs de cette thèse

Cette thèse a pour but de concevoir et de développer une plate-forme logicielle de pilotage de simulations scientifiques couplées et distribuées. Pour ce faire, nous nous appuierons sur la plate-forme EPSN qui permet le pilotage de simulations parallèles. Notre plate-forme devra être la plus générique possible afin d'offrir un service de pilotage à des simulations existantes sans avoir à les réécrire. Mais, elle devra également être la moins intrusive possible sur l'exécution du code. Nous allons donc devoir modéliser les simulations afin de pouvoir interagir de façon cohérente avec ces dernières. Par ailleurs, les mécanismes assurant la cohérence durant les interactions de pilotage devront être aussi le moins intrusifs possible. Les objectifs sont donc doubles.

Modéliser les simulations distribuées. La modélisation sera une partie importante de nos travaux. À partir d'une modélisation de haut niveau, nous pouvons exprimer les informations

minimum pour assurer la cohérence des traitements de pilotage et ce de façon indépendante des codes de simulations. Cette modélisation repose sur un modèle de description des simulations ainsi que sur un modèle de pilotage. Le modèle de description a pour rôle de décrire le flot d'exécution des simulations distribuées ainsi que les données utiles à l'analyse. Le modèle de pilotage quant à lui, permet de définir quand, comment et où interagir avec la simulation. Notre modélisation s'appuiera sur celle utilisée dans la plate-forme EPSN et sera étendue au cas des simulations distribuées.

Garantir la cohérence des traitements de pilotage. Au cours du processus de pilotage, on doit s'assurer que les traitements que nous effectuons sont cohérents en temps et en espace ; par là, on entend qu'ils doivent intervenir au même « *temps simulé* » et sur des plages de codes où les données sont accessibles (consultables ou modifiables). Dans le cas de simulations distribuées, cela peut s'avérer assez complexe car il faut commencer par définir ce qu'on appelle le même « *temps simulé* » entre les différents codes et surtout il faut pouvoir identifier les données correspondant à une même information. En effet, dans le cas de la visualisation « *en ligne* » des données nous voulons potentiellement voir les données des différents codes comme étant la même information. Pour assurer la cohérence dans les simulations couplées, nous utiliserons des algorithmes de planification qui ne synchronisent, *a priori*, pas les processus de la simulation. Nous devrons de plus nous assurer que la cohérence est conservée durant tous les traitements de pilotage. Pour la visualisation *en ligne* cela signifie qu'il faut assurer la cohérence des données depuis leur extraction des codes de simulation jusqu'à leur affichage.

Organisation de la thèse

Cette thèse est divisée en quatre chapitres. Le premier chapitre introduit les différentes notions utiles à la compréhension des travaux effectués. Il présente les bases concernant les simulations numériques, la visualisation scientifique et le couplage de code, puis il fait l'état de l'art sur les environnements de couplage et de pilotage de simulations. Le second chapitre rappelle le modèle abstrait de la plate-forme EPSN et décrit les évolutions apportées pour les simulations distribuées. Le troisième chapitre décrit comment le modèle abstrait de pilotage des simulations couplées a été implanté dans la plate-forme EPSN2 et également comment transformer une simulation en une simulation « *pilotable* ». Le dernier chapitre quant à lui présente l'évaluation expérimentale de la plate-forme EPSN2 afin de valider l'ensemble des travaux. Ce manuscrit de thèse se termine par une conclusion avec les perspectives de ce travail.

Chapitre 1

Présentation du problème, état de l'art et positionnement de notre approche

Sommaire

1.1 Simulations numériques parallèles distribuées . . . 8
1.1.1 La simulation numérique . . . 8
1.1.2 Simulations numériques distribuées . . . 9
1.1.2.1 Simulations M-SPMD . . . 9
1.1.2.2 Simulations Client/Serveur . . . 9
1.2 Visualisation scientifique . . . 11
1.2.1 Principes de base . . . 11
1.2.1.1 Le pipeline de visualisation . . . 12
1.2.1.2 Le pipeline graphique . . . 13
1.2.2 Techniques de visualisation parallèle . . . 13
1.2.2.1 Visualisation parallèle . . . 14
1.2.2.2 Rendu parallèle . . . 14
1.3 Les bases des environnements de couplage . . . 16
1.3.1 La couche de communication . . . 17
1.3.1.1 Couplage de codes et MPI . . . 17
1.3.1.2 Couplage de codes et CORBA . . . 18
1.3.2 La couche de redistribution . . . 19
1.3.3 Les modèles de haut niveau . . . 22
1.4 État de l'art sur le pilotage de simulations . . . 27
1.4.1 Introduction au pilotage de simulations . . . 27
1.4.2 Environnements de pilotage existants . . . 28
1.4.2.1 Caractéristiques des environnements de pilotage . . . 29
1.4.2.2 Environnements de pilotage existants . . . 30
1.4.2.3 Discussion . . . 36
1.4.3 Les limites des environnements existants . . . 36
1.5 Positionnement . . . 38
1.5.1 Modélisation des simulations couplées . . . 39
1.5.2 Cohérence des traitements . . . 39

1.1 Simulations numériques parallèles distribuées

Les simulations numériques sont devenues une partie importante dans l'étude de phénomènes physiques complexes. Elles permettent de compléter l'expérimentation soit en essayant de prédire les résultats, soit en la remplaçant dans le cas où l'expérimentation n'est pas accessible. Elle permet donc d'affiner la compréhension de ces phénomènes physiques.

Ces simulations numériques rentrent dans la catégorie du calcul haute performance ou HPC (High-Performance Computing). Ce sont généralement des simulations numériques parallèles, et ce afin de pouvoir exploiter pleinement les capacités des super-calculateurs actuels. Cela permet entre autre d'exécuter des simulations sur des cas tests plus gros, demandant trop de mémoire pour une machine seule, ou bien encore de s'exécuter plus rapidement. En effet, les super-calculateurs actuels, sont essentiellement des clusters (82% des calculateurs figurant au TOP 500 [133]).

1.1.1 La simulation numérique

Comme nous l'avons dit, une simulation numérique permet de simuler des phénomènes physiques complexes. Ainsi, pour un phénomène donné, il faut passer par plusieurs étapes afin de construire une simulation numérique : *modélisation*, *approximation* puis la *simulation* elle-même.

Il faut donc tout d'abord définir le modèle mathématique qui régit les lois physiques que l'on souhaite simuler. Ce modèle consiste généralement en un système d'équations aux dérivées partielles (EDP) décrivant l'évolution du phénomène physique au cours du temps. Par exemple, nous avons les équations de Navier-Stokes pour décrire l'écoulement de fluide compressible, ou bien l'équation de la chaleur pour décrire le phénomène de diffusion thermique. À ce modèle il faut adjoindre un domaine d'application (aile d'avion, molécule, *etc.*). Et pour finir, le modèle contient des conditions initiales et des conditions aux limites. Les conditions initiales représentent l'état initial du système, comme par exemple la vitesse initiale d'un fluide dans un conduit. Les conditions aux limites sont les conditions qui sont imposées sur les bords du domaine. Par exemple, la température qui est imposée sur les extrémités d'une barre métallique que l'on chauffe.

Une fois le système modélisé, il nous reste à résoudre ces équations. C'est là que l'informatique entre en action. Mais un ordinateur ne peut pas résoudre directement un système d'EDP. Par conséquent, il faut approximer les équations en les discrétisant avec des méthodes de résolution telles que les éléments finis, les différences finies ou les volumes finis. De plus, il faut également discrétiser le domaine sur lequel nous voulons effectuer les calculs. Ainsi nous allons par exemple, discrétiser une aile d'avion à l'aide d'un maillage afin de calculer les écoulements d'air à sa surface. Il est à noter que l'approximation est d'autant plus proche de la solution exacte que le pas de discrétisation est fin (le taille des éléments discret, la pas de temps, *etc.*). Ainsi, plus nous voulons nous rapprocher de la solution exacte et plus cela engendrera de calcul.

Une fois l'approximation des équations effectuée, il ne reste plus qu'à calculer la solution. Pour ce faire, nous allons écrire un code de simulation. Comme nous l'avons vu, plus nous voulons un résultat précis, plus la quantité de calcul sera importante et par conséquent plus les ressources de calcul (puissance de calcul, mémoire, *etc.*) devront être importantes. Nous nous tournons donc naturellement vers des simulations numériques parallèles afin d'exploiter pleinement la ressource de calcul fournie par les super-calculateurs. Ces simulations sont généralement des programmes itératifs dont la boucle principale représente l'évolution dans le temps du phénomène simulé.

1.1.2 Simulations numériques distribuées

Les simulations numériques calculent généralement un seul phénomène physique et donc elles ne résolvent qu'un seul système d'équations. Mais afin de simuler des phénomènes plus complexes et donc plus proche de la réalité, il faut pouvoir prendre en compte les interactions de phénomènes physiques différents. Ainsi, on doit coupler plusieurs modèles mathématiques, chacun représenté par un code, ce qui dérive assez naturellement sur un couplage de codes. Les simulations ne sont donc plus de simples « *codes* », mais des assemblages de codes, le plus souvent parallèles, chaque code d'un tel assemblage gérant un système d'équations et/ou un domaine de calcul différent. Ainsi nous en arrivons à des simulations multi-physiques et/ou multi-échelles. Ces simulations peuvent être construites à partir de codes développés par des experts de chaque domaine (*legacy codes*[1]), ce qui permet d'avoir un haut degré d'expertise pour chaque modèle qui entre en jeu dans le couplage. Cela permet également une plus grande modularité car chaque code peut continuer d'être développé, maintenu et testé séparément.

Ces simulations sont généralement construites suivant des schémas d'assemblage bien précis. Dans la suite de cette thèse nous allons en retenir deux, qui sont les deux types de couplage les plus répandus dans les simulations HPC : les simulations de type M-SPMD (M-SPMD) pour « *Multiple - Simple Program Multiple Data* » et Client/Serveur.

1.1.2.1 Simulations M-SPMD

Les simulations M-SPMD sont généralement des simulations construites à partir de plusieurs codes SPMD dont la structure globale est similaire, ils ont tous leur propre boucle principale et c'est dans cette boucle que l'on peut noter des similarités dans les tâches effectuées. Dans ce type de couplage, chaque code gère une physique ou un domaine et échange régulièrement des données avec les autres codes, en début/fin de simulation pour l'initialisation et la production de diagnostiques, et au cours de la boucle principale afin de prendre en compte des paramètres venant des autres physiques ou des autres domaines.

Un bon exemple est la simulation réalisé par la LibMultiScale [5] qui est une simulation M-SPMD. Cette simulation couple un code de dynamique moléculaire à un code d'éléments finis. Il s'agit d'une simulation multi-échelles où chacun des sous-codes gère son propre domaine. Les codes ont tous leur propre boucle en temps. À la fin de chaque itération, les codes s'échangent des informations sur la frontière commune des domaines. L'information que l'on souhaitera visualiser pour cette simulation est le déplacement des atomes (voir figure 1.1) et cela s'exprime plus ou moins différemment suivant le code. Ainsi dans le code de dynamique moléculaire, on peut avoir à tout moment les positions actuelles des atomes, alors que dans le code d'éléments finis, on aura directement l'information du déplacement par rapport à la position d'origine.

1.1.2.2 Simulations Client/Serveur

Les simulations Client/Serveur sont quant à elles des simulations basées sur un code client qui fait appel à des serveurs pour résoudre certains sous-problèmes. Le code client peut être de plusieurs types. Il peut être soit un code ne servant qu'à coupler les sous-codes serveurs, soit un code à part entière faisant appel à des serveurs pour résoudre un problème donné. Dans tous les cas, c'est le code client qui « *gère une boucle principale* » et qui ordonnance les appels des différents serveurs.

1. Par la suite pour désigner ces codes de simulation, nous parlerons de code existant.

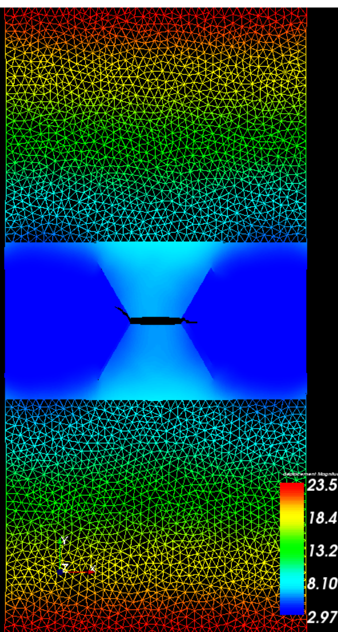


Figure 1.1 – Exemple de propagation d'une fissure simulée avec le LibMultiScale (M-SPMD)

Nous avons le couplage des codes de simulation DLPoly [32] et Siesta [127] qui sont respectivement des codes de dynamique moléculaire et de mécanique quantique, communément appelé couplage QM/MM (voire figure 1.2). Ce type de couplage permet d'allier la vitesse de calcul des méthodes de mécanique moléculaire (MM) à la précision des méthodes de mécanique quantique (QM). Dans ce genre de couplage Client/Serveur, le code MM joue le rôle de client, il est composé d'une boucle en temps au cours de laquelle il calcule les forces d'interaction entre les différents atomes en fonction des charges de ces derniers. Pour pouvoir calculer ces forces, il faut que le code de mécanique quantique calcule la densité de charge de la zone « *quantique* ». Dans ce couplage nous chercherons à visualiser la charge des atomes, c'est-à-dire la densité de charge dans le cas du code QM et les charges des atomes dans le cas du code MM.

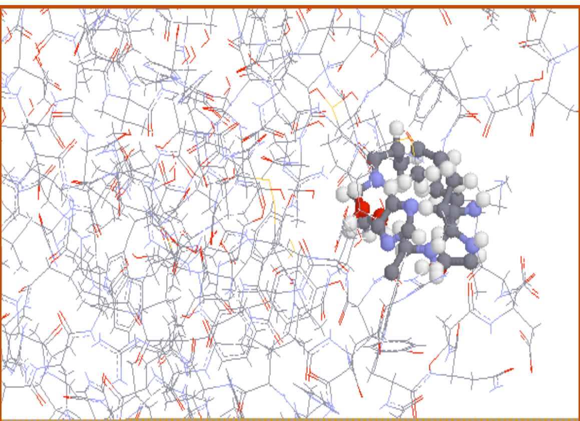


Figure 1.2 – Exemple de couplage QM (à droite) / MM (à gauche)

Dans ces types de couplage, nous pouvons noter qu'il y a toujours une unique boucle prin-

cipale. Dans le cas des simulations M-SPMD, nous pouvons l'exprimer de façon implicite car chaque code a une boucle en temps correspondant à une même évolution globale. Ainsi, en décrivant les différents codes du couplage à un niveau assez élevé, pour gommer leurs différences, nous pouvons expliciter cette boucle en temps unique au cours de laquelle les codes s'exécutent simultanément. Pour ce qui est des simulations Client/Serveur, la boucle principale est explicite dans le code client et c'est ce code qui fait des appels à distance sur des codes serveurs.

En ce qui concerne les données des simulations couplées dans ces deux types de couplage, nous avons des données différentes sur les codes, que ce soit en terme de support (grilles, maillages, points, *etc.*.) ou de variables associées à ce support (*e.g.*charges des atomes). Mais au niveau de la visualisation, seules certaines propriétés nous intéressent et elles peuvent généralement être exprimées en fonction des différents données des codes.

Dans le contexte de la simulation numérique il y a un besoin d'analyser les résultats produits par les simulations, que ce soit en « *temps-réel* » ou en post-traitement. Nous allons donc présenter brièvement en quoi consiste la visualisation scientifique.

1.2 Visualisation scientifique

La visualisation scientifique joue un rôle important dans la phase d'analyse des données scientifiques produites par des simulations numériques. La visualisation scientifique est définie comme le processus de conversion des données numériques en une représentation graphique facile à interpréter. Dans cette section, nous allons présenter quelques notions de base sur la visualisation scientifique, ainsi que sur les techniques de rendu parallèle. Dans le contexte du pilotage de simulation ces techniques peuvent être utilisées afin d'avoir une visualisation « *en-ligne* » des données produites par une simulation.

1.2.1 Principes de base

La visualisation scientifique s'appuie sur un principe de *pipeline* de traitement des données scientifiques. Les *pipelines* de visualisation scientifique peuvent être vus comme deux *pipelines*, un *pipeline de visualisation* dont le rôle est de transformer les données scientifiques en primitives graphiques, et un *pipeline graphique* dont le rôle est de transformer ces primitives graphiques en images 2D affichables à l'écran comme illustré par la figure 1.3. En règle générale, la visualisation scientifique intervient comme une étape de post-traitement des données d'une simulation numérique. En pratique, les simulations vont générer des fichiers de données qui seront traités par des logiciels de visualisation. Parmi ces logiciels, on pourra citer IRIS Explorer[48], AVS [7], OpenDX [67], Ensight [37], Paraview [110], VisIt [140] ou encore VTK [141].

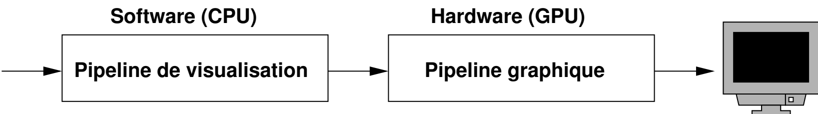


Figure 1.3 – Pipeline de visualisation scientifique.

La plupart de ces produits permettent de développer des *plugins* afin de les étendre suivant des besoins particuliers, essentiellement pour traiter d'autres formats de fichier d'entrée.

1.2.1.1 Le pipeline de visualisation

La plupart des systèmes de visualisation sont basés sur un paradigme *data-flow*. Dans ce paradigme, le processus de traitement des données est décomposé en unités d'exécution, ou *modules*. Ces *modules* possèdent des entrées et des sorties, ou *ports*. Ces ports permettent de connecter entre eux les modules. Ainsi on peut connecter les *ports* de sorties d'un *module* sur les *ports* d'entrées d'un autre. Dans ce cas lorsque le premier *module* aura fini son traitement, il enverra les données qu'il a généré au(x) *module(s)* au(x)quel(s) il est connecté. L'ensemble des *modules* inter-connectés forme un graphe *data-flow* ou *pipeline*. Les données sont envoyées dans le premier *module* du graphe et traitées par les différents modules jusqu'à arriver au dernier *module*. L'exécution du *pipeline*, ou la mise à jour, fait toujours suite à une action. On distingue différents types de *pipeline*, suivant le mécanisme de mise à jour utilisé. Ainsi on a les *pipelines* dirigés par des évènements (*event-driven*) ou dirigés par des « *demandes* » (*demand-driven*). Dans le premier cas, l'exécution est commandée par un évènement arrivant à l'entrée du pipeline, comme une modification des données. Dans le second cas, la mise à jour fait suite à une demande venant de la sortie du pipeline. Dans ce dernier cas la mise à jour n'est pas forcement complète, c'est-à-dire que la mise à jour débute dans le premier module dont les entrées ont changé. Dans ces pipelines, on distingue trois catégories de modules illustrées par la figure 1.4 : les *sources*, les *filtres* et les *mappers*.

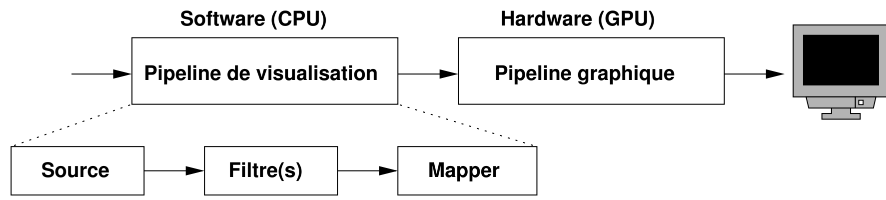


FIGURE 1.4 – Composition du pipeline de visualisation.

Source – Les *sources* sont les modules d'entrées du pipeline de visualisation. Il existe différents types de sources, soit des sources d'import, soit des sources dites procédurales. Les sources d'imports permettent de lire des fichiers de données de différents formats, afin de les transformer en données compréhensibles par le système de visualisation. Les sources procédurales, quant à elles, construisent les données à partir d'une procédure, comme par exemple des sphères, des cônes, *etc.*.

Filtre – Les *filtres* traitent les données issues des *sources* afin d'en extraire les données pertinentes pour la visualisation. On peut potentiellement combiner plusieurs filtres pour arriver au résultat souhaité. Pour ce faire, les filtres implantent des algorithmes plus ou moins complexes, comme par exemples des plans de coupe, des iso-surfaces, des lignes de courant, *etc.*.

Mapper – Les *mappers* servent de lien entre le *pipelines de visualisation* et les *pipelines graphiques*. Ils transforment les données en sortie des filtres en objets graphiques 3D (points, lignes, polygones). Ces objets 3D sont envoyés au pipeline graphique afin d'être transformés en image 2D.

1.2.1.2 Le pipeline graphique

Les données sortant du *pipeline de visualisation* sont donc des objets 3D, ces objets doivent encore être traités afin d'arriver à une image 2D affichable sur un écran. Afin de produire un « *rendu* », il faut définir une scène complète, c'est-à-dire avec des sources de lumière et une caméra en plus des objets 3D. Le pipeline graphique va donc traiter cette scène et afficher l'image résultante sur un écran. Il est à noter que ce pipeline est généralement exécuté sur une carte graphique bénéficiant d'accélération matérielle. Ce pipeline est également divisé en trois parties, comme le montre la figure 1.5.

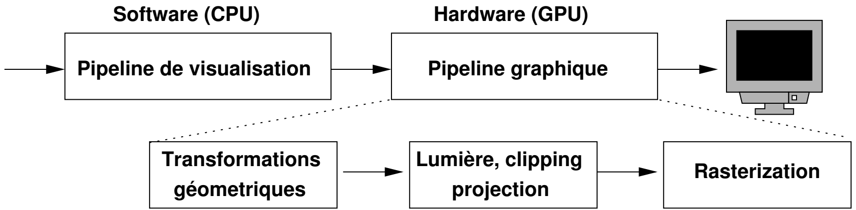


FIGURE 1.5 – Composition du pipeline graphique.

Transformation géométrique – Cette partie du pipeline consiste à appliquer les transformations géométriques (translation, rotation, mise à l'échelle) sur les objets 3D, afin d'obtenir les coordonnées dans le repère de la scène 3D. On passe de l'espace objet à l'espace monde.

Application des attributs – Cette étape consiste en l'application de l'illumination. Cela revient à déterminer les couleurs des différents sommets des polygones. La détermination de la couleur va dépendre de l'algorithme de *shading* (*Flat*, *Gouraud*, *Phong*). Le *flat shading*, consiste à donner une couleur unique aux polygones. Le *Gouraud shading* consiste à déterminer une couleur par sommets et à effectuer une interpolation linéaire sur les polygones couvrant la surface du polygone. Et enfin, le *Phong shading* consiste tout comme dans le modèle *Gouraud* en une interpolation linéaire, mais ici ce sont les normales aux sommets qui sont interpolées [2]. C'est également lors de cette étape que sont appliqués les textures sur les polygones. Et pour finir, c'est également à ce moment que les coordonnées des objets sont passées de l'espace monde à l'espace de projection. Cela permet d'effectuer le *clipping*, c'est-à-dire éliminer tous les polygones qui ne sont pas dans le champs de la caméra.

Rasterization – Cette étape est la dernière étape du *pipeline graphique*. Elle consiste à calculer l'image 2D finale. Pour ce faire, on utilise un algorithme de *Z-buffer* (tampon de profondeur). Cet algorithme permet de déterminer quels sont les polygones visibles en fonction de leurs profondeurs dans la scène. C'est également durant cette étape que la transparence des objets est prise en compte via l'α-buffer (tampon d'opacité). Ce tampon permet « *d'accumuler* » les couleurs des différents polygones qui se superposent.

1.2.2 Techniques de visualisation parallèle

Dans le cas où les données à visualiser sont trop volumineuses ou que les traitements à effectuer sont trop lourds pour un processeur, les techniques séquentielles de visualisation et de

2. les vecteurs normaux à la surface permettent de déterminer finement le modèle de réflexion de la lumière afin de déterminer la couleur finale des *pixels*

rendu classiques ne sont plus suffisantes. Dans ce cas, on peut avoir recours à des techniques de visualisation parallèle et/ou de rendu parallèle. Cela permet de répartir les données et les traitements sur plusieurs nœuds de calcul et/ou de rendu.

1.2.2.1 Visualisation parallèle

Il existe plusieurs méthodes pour paralléliser les traitements de visualisation scientifique. Ainsi, on trouve des travaux sur le calcul d'iso-surface [59] en parallèle, ou bien encore sur la simplification géométrique pour réduire le niveau de détail des objets à visualiser [61, 62, 64]. Nous allons détailler plus particulièrement les techniques de visualisation parallèle utilisées dans VTK. En effet, la bibliothèque VTK intègre de nombreux travaux autour de la visualisation scientifique parallèle [1, 44, 82, 83] qui sont une référence de nos jours. Le principe de parallélisation dans VTK consiste essentiellement à découper les données et à les traiter via un même pipeline de visualisation scientifique répliqué dans des processus séparés. De plus, les différents modules du pipeline peuvent être exécutés en parallèle sur des données indépendantes. La figure 1.6 montre ainsi un pipeline de visualisation répliqué sur quatre noeuds de calcul.

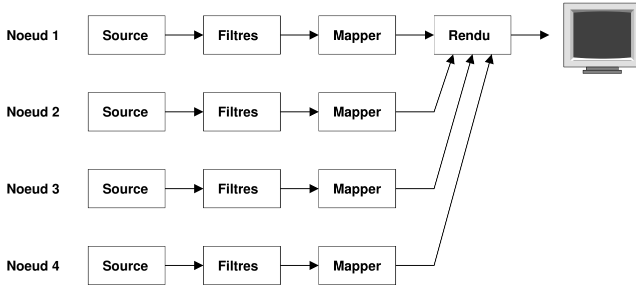


FIGURE 1.6 – Pipeline de visualisation parallèle.

Dans ce type de parallélisation, il y a potentiellement un goulot d'étranglement lors de la phase de rendu. Ainsi, si le traitement des données ne réduit pas le volume de données à visualiser, la phase de rendu séquentiel peut être très coûteuse. Par conséquent, on va également vouloir paralléliser l'étape de rendu.

1.2.2.2 Rendu parallèle

Dans la parallélisation de l'étape de rendu, nous dupliquons le rendu sur tous les nœuds de traitement tel que présenté sur le figure 1.7. Ainsi, nous répliquons le pipeline de rendu à priori autant de fois que le pipeline de visualisation. Par ailleurs, les techniques de rendu parallèle permettent également de visualiser les données sur plusieurs écrans. Pour ce faire, il existe différentes manières de procéder. Les plus courantes, que nous allons à présent détailler, sont les techniques de *sort-first* et de *sort-last*.

La technique de *sort-first* consiste à trier les primitives graphiques à la sortie du pipeline de visualisation. Pour ce faire, chaque processus de rendu est responsable d'une portion de l'image finale. Les primitives graphiques sont donc envoyées aux processus qui sont responsables de la zone spatiale dans laquelle elles seront affichées. À la base, les données sont reparties dans les pipelines de visualisation de façon arbitraire. Pour les envoyer sur les bons processus, une phase de redistribution des polygones sera donc nécessaire. Cette redistribution passe par une étape de tri des polygones, suivi d'une étape de communication. La figure 1.8 illustre un calcul de visualisation sur 5 processus, et un rendu en *sort-first* sur 3 écrans. Sur cette figure, on peut

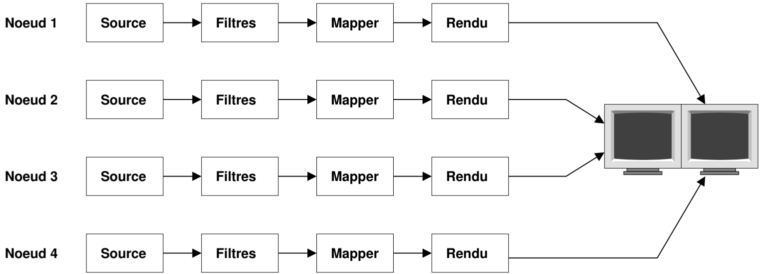


FIGURE 1.7 – Pipeline de visualisation parallèle.

noter que l'équilibrage du rendu dépend fortement de l'affichage final. Ainsi, si un écran devait afficher beaucoup plus de polygones que les autres, le processus associé aurait plus de travail que les autres.

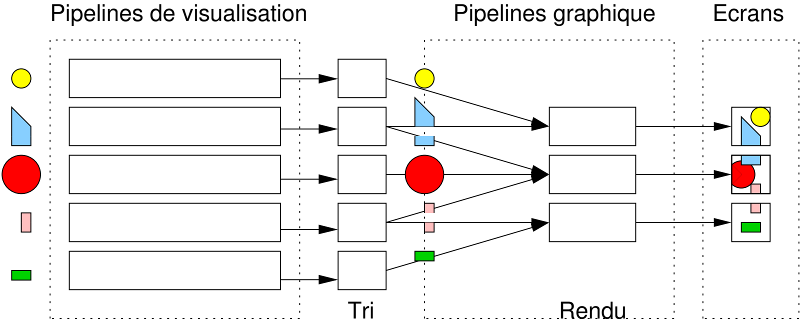


FIGURE 1.8 – Rendu parallèle en *sort-first* et affichage sur mur de 3 écrans.

Dans la technique de *sort-last*, chaque pipeline construit une image partielle de la scène avec la même résolution que l'image finale. Ces images sont ensuite composées afin d'obtenir l'image finale. Cette composition est faite en comparant les *Z-buffers* partiels à la sortie de tous les pipelines de rendu. Cette méthode a l'avantage d'engendrer un volume de communication plus faible que le *sort-first*, vu que seul des images 2D sont échangées lors de la phase de composition. De plus, le calcul du rendu peut être fait sur plus de processus que l'affichage. La même configuration que sur la figure 1.8 est reprise sur le figure 1.9 afin d'illustrer la méthode du *sort-last*.

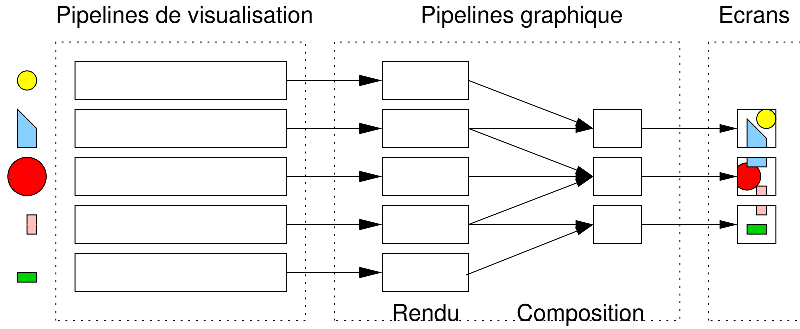


FIGURE 1.9 – Rendu parallèle en *sort-last* et affichage sur mur de 3 écrans.

Les techniques de *sort-first* sont mises en œuvre dans certains environnements tel que Chromium [22, 65] ou DCV (Deep Computing Visualization) [66]. Ces logiciels interceptent les appels à la bibliothèque OpenGL pour produire un rendu parallèle . En effet, cela ne demande aucune modification des logiciels de visualisation, tout du moins dans le cas où ils sont séquentiels. Pour ce qui est du *sort-last*, on peut citer Chromium, VTK [94] ou bien encore Ice-T [95, 144]. Ice-T est utilisé dans Paraview afin de faire du rendu en *sort-last* sur mur d'écrans.

Nous avons vu les différentes manières d'effectuer de la visualisation scientifique, ainsi que les différents types de simulations qui nous intéresseront dans ces travaux. Nous avons donc introduit le processus de simulation et d'analyse. Nous allons à présent voir plus en détail comment sont construits de telles simulations. Pour ce faire, nous pouvons noter que les simulations qu'elles soient de type M-SPMD ou Client/Serveur, font appel à des environnements de logiciels permettant de les construire.

1.3 Les bases des environnements de couplage

Le couplage de simulation est un processus complexe faisant appel à différents types d'outils logiciels qui permettent sa mise en place. Ainsi nous allons voir que le couplage entre deux simulations peut être fait à différents niveaux et de différentes manières. En effet pour coupler plusieurs codes, il faut pouvoir communiquer des données de l'un à l'autre. Mais, dans le cas où ces codes sont parallèles, cela peut s'avérer difficile car chaque code aura sa propre distribution des données ainsi que des nombres de processus potentiellement différents. Il nous faudra donc faire appel à des mécanismes de redistribution de données afin de définir les schémas de communication d'un code à un autre. Avec ces deux niveaux, communication et redistribution, nous pouvons déjà coupler des codes de façon « *ad-hoc* », c'est-à-dire que les développeurs peuvent écrire le couplage « *à la main* » pour chaque simulation. Les environnements de redistribution permettent d'introduire un début d'abstraction par rapport aux codes de simulation, en décrivant les données et leurs distributions. Mais pour rendre les environnements plus génériques et réutilisables, il est a priori nécessaire de modéliser les simulations avec des descriptions de haut niveau des données et du flot d'exécution. Les environnements de couplage sont définis à partir d'une telle modélisation. De plus, ces environnements s'appuieront sur les couches de communication et de redistribution.

Ces différents niveaux d'environnements permettent donc de définir des couplages de simulations de façon plus ou moins générique comme le montre la figure 1.10. Les différentes couches permettent donc de catégoriser les couplages. La couche de communication est la couche la plus basse et elle demande par conséquent le plus de travail de la part du développeur car pour chaque couplage, il faut redéterminer tous les schémas de communication et refaire la gestion des données à envoyer/recevoir. La couche de redistribution nécessite moins de modifications car les environnements permettent de décrire les données et leur distribution et ils en déduisent les schémas de communication. Mais il faut tout de même refaire le travail de description pour chaque couplage. La couche de modélisation permet de réutiliser les codes des simulations qui ont été modélisés dans différents couplages.

En plus de pouvoir limiter le travail des développeurs, la modélisation permet de piloter les simulations. En effet le pilotage de simulations peut être vu comme une application du couplage de code. La modélisation de haut niveau permet de gérer les données, mais également le flot d'exécution.

Dans la suite de cette section, nous allons détailler ces différentes couches et voir comment elles interviennent dans le processus de couplage de codes.

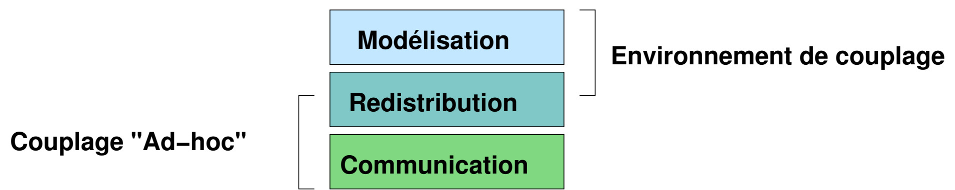


Figure 1.10 – Couches logicielles du couplage de code

1.3.1 La couche de communication

Afin de pouvoir coupler des codes de simulation entre eux, il nous faut un moyen de les faire communiquer. Pour cela il existe beaucoup de méthodes, plus ou moins évoluées, allant des sockets TCP/IP à l'appel de méthodes à distance (ou RMI), en passant par les bibliothèques d'échange de messages. Les simulations parallèles sont majoritairement basées sur des bibliothèques d'échange de message telle que MPI (Message Passing Interface) [57], le standard *de facto* de la programmation parallèle.

Les simulations parallèles sont généralement exécutées sur des plates-formes homogènes et ainsi il n'y a pas de problème d'interopérabilité, ce qui dans le cas des codes couplés n'est plus forcément le cas. En effet les simulations distribuées sont généralement distribuées sur plusieurs plates-formes de calcul, ou *clusters*. L'homogénéité n'est donc plus assurée entre les différents clusters. La plupart du temps les plates-formes multi-clusters sont constituées de clusters avec des réseaux haute performance entre les nœuds de calcul tels que Infiniband ou Myrinet, alors que les interconnexions entre les clusters sont souvent des réseaux Ethernet (Gigabit Ethernet ou 10 Gigabit Ethernet). La problématique de la communication n'est donc plus limitée à un simple échange de données. Dans le cas du couplage du code, il faut prendre en compte l'hétérogénéité qui existe entre les différents codes (langage de programmation) et également entre les clusters (encodage des données, système d'exploitation, réseaux) sur lesquelles les codes de simulation sont exécutés. Pour ce faire, il existe plusieurs solutions. Nous allons en présenter quelques unes basées sur MPI [69], ainsi que sur CORBA (Common Object Request Broker Architecture) [107].

1.3.1.1 Couplage de codes et MPI

MPI étant un standard reconnu pour le développement d'applications parallèles, il est naturel de vouloir l'utiliser pour développer des applications distribuées. Ces nouvelles applications n'étant rien d'autre que le couplage de simulations parallèles déjà existantes. MPI va donc servir soit pour effectuer des couplages « *ad-hoc* », soit pour construire des environnements de couplage.

Pour coupler des codes avec MPI, il y a plusieurs solutions suivant la version de MPI utilisée. En *MPI-1* [69], la plus simple est de modifier les codes afin de changer le communicateur *MPI_COMM_WORLD* par des communicateurs locaux aux codes. *MPI_COMM_WORLD* devient donc le communicateur commun aux codes. Cela implique de pouvoir lancer les codes de la simulation via un lanceur unique et ce sur un calculateur homogène.

Avec la spécification *MPI-2* [91], nous pouvons construire des couplages de façon plus dynamique. Le mode client/serveur introduit dans la spécification de *MPI-2* permet de lancer les codes séparément et de les connecter par la suite via les méthodes *connect/accept*. Ces codes peuvent également être lancés depuis un processus MPI via la méthode *spawn*.

Le principal problème concernant MPI est que les spécifications ne précisent rien quant à l'hétérogénéité, que ce soit au niveau des réseaux ou des langages de programmation. Or lorsque l'on parle de couplage de codes, on s'attend à pouvoir exécuter des codes sur des clusters différents

et potentiellement hétérogènes. Cette hétérogénéité peut être de plusieurs types, architecture, réseaux. Il existe plusieurs solutions pour contourner ce problème. Il y a des environnements tels que PACX-MPI [9] ou PVMPI [35] qui permettent d'utiliser des implantations performantes de MPI sur chaque cluster et de communiquer entre les clusters par le biais de TCP/IP ou de PVM. Ces solutions permettent d'utiliser les implantations de MPI fournies avec les clusters. Cette solution fait de plus en plus place à une approche basée sur des implantations de MPI plus génériques, telle que MPICH-Madeleine [89, 90], OpenMPI [55, 109], ou plus récemment MPICH-2 [80] qui permettent d'utiliser plusieurs types de réseaux différents de façon simultanée. Il est à noter que OpenMPI et MPICH-2 implantent la spécification *MPI-2*.

1.3.1.2 Couplage de codes et CORBA

Une autre méthode utilisée dans le couplage de code est basée sur des communications via des appels de méthodes à distance. Ce type de communication implique un modèle de programmation Client/Serveur. On trouve différentes implantations de ce type de communication telles que les Sun RPC (Remote Procedure Call) [93], GridRPC [126], Microsoft DCOM (Distributed Component Object Model) [92], Java RMI (Remote Methode Invocation) [143] ou CORBA (Common Object Request Broker Architecture) [105, 107]. Dans cette partie, nous allons détailler un peu plus CORBA qui est souvent utilisé dans le cadre du couplage de simulations.

CORBA est un standard défini par l'OMG (Object Management Group) [103] qui permet d'écrire du code portable, c'est-à-dire pouvant être écrit dans différents langages et pour différents systèmes d'exploitation. Il permet de définir des programmes à base d'objets distribués pouvant être exécutés sur des machines hétérogènes.

Les objets distribués sont des objets permettant d'effectuer des appels de méthodes à distance. Ces appels se font de manière transparente au niveau du langage de programmation. Pour cela, le mécanisme d'appel de méthodes à distance passe par des souches et des squelettes, qui sont des descriptions locales de l'objet et qui sérialisent les appels de méthodes pour les envoyer sur le réseau, comme le montre la figure 1.11.

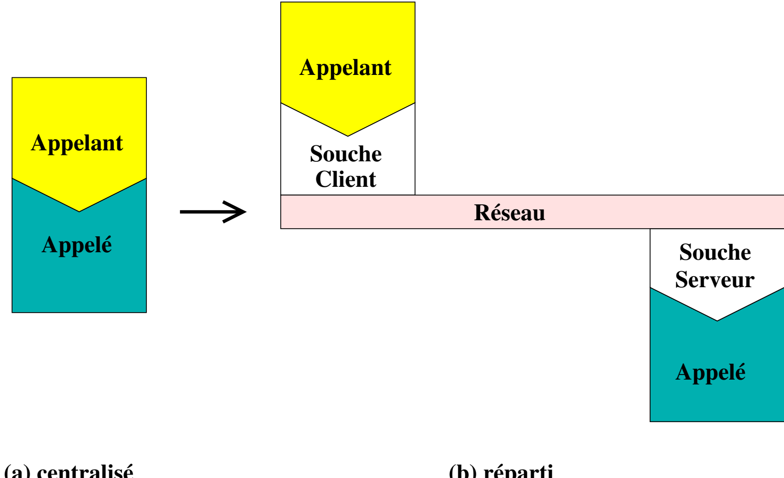


FIGURE 1.11 – Mécanisme d'appel de méthodes à distance

Dans le cadre de CORBA, les objets sont décrits grâce à un langage de description d'interface, l'IDL (Interface Definition Language), et les appels à distance sont faits par le biais de l'ORB (Object Request Broker) qui lui même s'appuie sur le GIOP (General Inter-ORB Protocol). Le

GIOP est une couche réseau virtuelle pouvant se décliner sous plusieurs implantations différentes, IIOP (Internet Inter-ORB Protocol) étant la plus commune basée sur TCP/IP.

Le fait que CORBA soit prévu pour fonctionner sur des plate-formes hétérogènes et qu'il soit dynamique, permet dans le cadre du couplage de palier aux manques d'une bibliothèque de communication telle que MPI-1. Ainsi, les programmes parallèles utilisent MPI localement pour les communications intra-codes et CORBA pour les communications inter-codes. La principale difficulté de cette approche est que nativement CORBA n'est pas prévu pour le parallélisme, que ce soit pour le déploiement de codes développés en CORBA ou pour la description des données. C'est pour cela qu'il y a des propositions d'extensions parallèles de CORBA qui permettent le couplage de codes parallèles, comme Pardis [75], Data Parallel Object [104] et PaCO++ [30, 114]. Pardis est basé sur l'introduction de nouveaux types de données dans la description IDL. Data Parallel Object est une spécification introduite par l'OMG nécessitant une modification de l'ORB afin d'avoir une architecture bas niveau parallèle de CORBA. PaCO++, quant à lui, introduit une description du parallélisme dans un fichier séparé afin de ne pas modifier l'IDL, ou l'ORB. Mais toutes ces approches restent bien souvent des prototypes, ou des preuves de faisabilité.

Ces deux moyens de définir les communications permettent donc de coupler des codes. Ces couplages se font souvent à partir de codes existants. Dans ce cas, le couplage de simulation est plus simple à mettre en œuvre à l'aide de MPI car les codes de simulation parallèles utilisent généralement déjà cette bibliothèque. Le couplage de code à l'aide de CORBA nécessite plus de modifications du code initial afin de l'adapter au concept d'objets distribués. En contrepartie, CORBA permet d'exécuter des codes couplés sur des plate-formes hétérogènes, ce qui n'est pas aussi aisé avec MPI.

1.3.2 La couche de redistribution

Une fois la couche de communication choisie, on souhaite échanger des données entre les codes. Or les codes étant parallèles, les données sont potentiellement distribuées sur les codes. Par conséquent, l'échange de données peut être une étape complexe nécessitant de redistribuer les données, c'est-à-dire qu'il faut passer d'une distribution des données à une autre. D'un code à un autre les données peuvent être réparties sur un nombre de processus différent, ou bien elles peuvent être décomposées différemment. Par exemple, dans le cas d'une simulation couplée traitant des grilles uniformes de données, celles-ci peuvent être décomposées en colonne dans un code et en ligne dans un autre. Il peut aussi être nécessaire d'effectuer des interpolations pour passer d'un support à un autre (*e.g.* passage d'une grille structurée à un maillage non structuré), ou d'une échelle à une autre. Le code se chargeant de cette étape peut être très complexe et devoir le réécrire pour chaque couplage peut devenir très coûteux. On s'orientera donc vers des intergiciels se chargeant de la redistribution des données.

Parmi ces intergiciels nous trouvons InterComm[84, 129], MpCCI[73, 96], Roccom [72, 125], MCT [71, 88], RedGRID [41], CumulVS [51, 78], PAWS [8, 74], CCA M×N [13], PRMI [29]. Nous allons détailler quelques unes de ces bibliothèques, afin de mieux comprendre en quoi elles consistent. Certaines d'entre elles servent à coupler directement des codes, d'autres sont utilisées pour construire des environnements de couplage.

InterComm [68, 84, 129], anciennement Meta-Chaos [34, 120], est un environnement de redistribution développé pour coupler des codes de simulation. Cet environnement est utilisé par exemple dans CCA afin de définir un composant de redistribution [13], ou bien encore dans une plate-forme CISM (Center for Integrated Space Weather) [54]. Cet environnement introduit le

principe de linéarisation. Cela consiste à ordonner les éléments d'un objet source et d'un objet destination afin de pouvoir les mettre en correspondance de façon implicite (voir figure 1.12). Afin d'utiliser InterComm, il faut décrire les données que l'on souhaite échanger. Les données

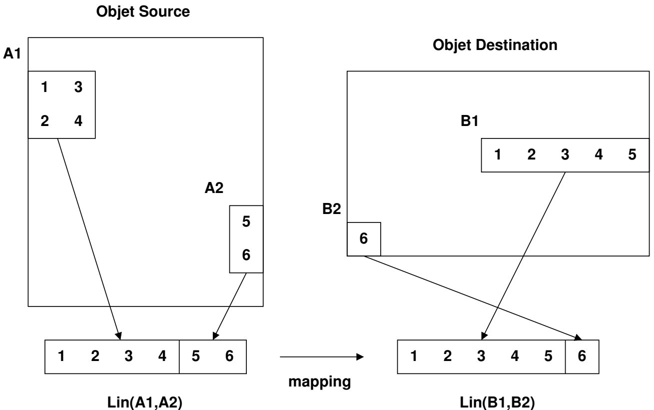


FIGURE 1.12 – Principe de linéarisation introduit par InterComm sur des grilles structurées 2D

que l'on peut décrire sont des *blocks* de tableaux de données, qui peuvent être distribués de façon régulière ou irrégulière entre les processus des codes. On peut également linéariser d'autres sortes d'objets comme des arbres, des maillages, *etc.*. Cette approche est très générique mais très coûteuse, car les données décrivant les objets sont potentiellement de la même taille que les données elles-mêmes. Pour finir, les transferts de données dans InterComm sont basés sur la bibliothèque PVM.

MpCCI (Mesh-based parallel Code Coupling Interface) [73, 96] est une bibliothèque de redistribution permettant le couplage de codes parallèles à base de maillages. Elle est essentiellement utilisée dans le cadre de couplages fluide/structure. Cette bibliothèque permet de décrire des maillages réguliers ou irréguliers et de les échanger entre différents codes via MPI. MpCCI inclut des algorithmes d'interpolation afin de pouvoir transférer des maillages qui ne coïncident pas. La figure 1.13 montre un échange de données lors d'un couplage de codes fluide/structure. On peut

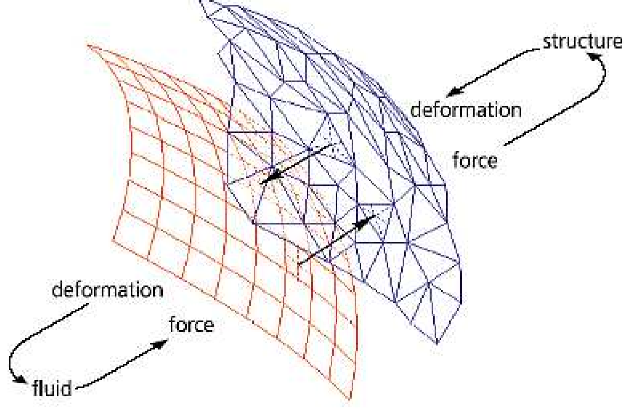


FIGURE 1.13 – Échange de données entre deux maillages ne coïncidant pas

noter que les échanges se font à partir d'une grille structurée vers un maillage non structuré.

La valeur du degré de liberté étant présente au centre des éléments du maillage et de la grille, il faut interpoler la donnée (ici la force et la déformation) au centre lors d'un transfert, car les éléments de la grille ne coïncident pas exactement avec ceux du maillage.

Roccom (Rocstar Component Object Manager) [72, 125] est une bibliothèque de redistribution utilisée dans le cadre du couplage d'applications multi-physiques pour des simulations de moteurs de fusée à propulsion solide. Cette bibliothèque est basée sur un concept de composants logiciels. Avec cette bibliothèque, chaque code d'une simulation est vu comme un composant qui peut enregistrer des fenêtres (*windows*). Ces *windows* encapsulent les données d'un code (voir figure 1.14). Ces données sont des maillages avec des variables qui peuvent être associées aux nœuds ou aux cellules du maillage. Dans le cas de codes parallèles, les *windows* peuvent être découpées en carreaux (*panes*), un *pane* étant associé à un processus. Les *windows* sont des objets distribués accessibles depuis tous les codes d'une simulation. Les codes d'une simulation peuvent donc s'échanger des données via ces *windows*. Les échanges se font via la bibliothèque de communications MPI.

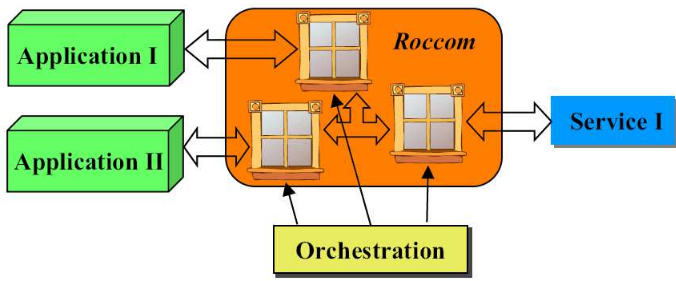


FIGURE 1.14 – Illustration schématique des *windows* et des *panes* dans Roccom

MCT (Model Coupling Toolkit) [71, 88] est une bibliothèque qui étend MPI afin de pouvoir redistribuer des grilles structurées entre des codes de simulation. MCT est utilisé dans le coupleur de l'environnement CCSM afin de coupler les modules d'océan, d'atmosphère, de glace et de terre [70]. Dans MCT, la distribution des données est décrite par la structure `GlobalSegmentMap`. Cette structure contient l'ensemble des éléments contigus d'un processus (ou *AttributeVector*). À partir de ces `GlobalSegmentMap`, MCT peut déterminer un *router* qui sert à ordonnancer les communications entre les différents processus. De plus MCT offre la possibilité d'effectuer des traitements parallèles tels que des interpolations sur les flux de données qui transitent via un *router*.

RedGRID [41, 122] est une bibliothèque de redistribution de données découplée d'un domaine scientifique particulier. Par conséquent, elle ne se limite pas à un seul type de données. Elle permet la redistribution de tableaux multi-dimentionnels, de particules, et de maillages. RedGRID est en fait divisé en trois couches logicielles, une couche de description symbolique des données (distribution et stockage), une couche de redistribution symbolique et une couche de communication. La couche de redistribution symbolique contient différents algorithmes de redistribution en fonction du type de données. La couche de communication quant à elle s'appuie sur CORBA, mais il existe également un prototype basé sur MPI. La figure 1.15 résume les différentes possibilités offertes par cette bibliothèque.

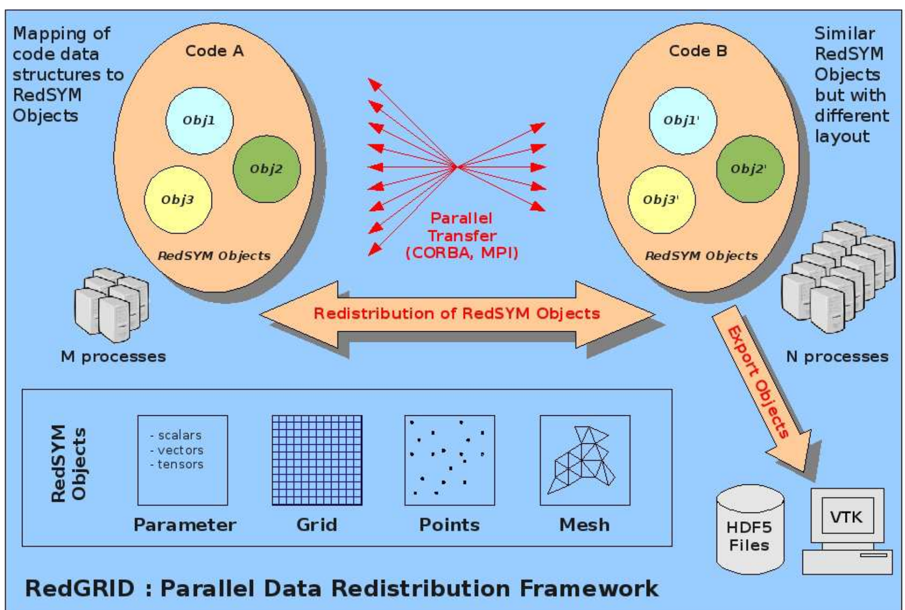


FIGURE 1.15 – Vue d'ensemble de la bibliothèque de redistribution RedGRID

Toutes les bibliothèques décrites dans cette section permettent de faire des couplages de simulations toujours de façon « *ad-hoc* », mais elles nous donnent tout de même une vision de plus haut niveau qu'en utilisant les bibliothèques de communication. La plupart de ces bibliothèques servent d'ailleurs dans des environnements de couplage plus complets incluant une modélisation de la simulation. Nous allons donc maintenant présenter ces différents environnements.

1.3.3 Les modèles de haut niveau

Pour pouvoir construire des couplages plus génériques, il faut pouvoir modéliser les simulations intervenant dans le couplage. Ainsi il devient possible de construire les couplages d'après les modèles et donc de remplacer un code dans un couplage par un autre code ayant une modélisation similaire. La plupart des environnements de couplage ont une modélisation basée sur la notion de composant. Nous allons donc commencer cette section par une description de ce que sont les composants logiciels.

Les composants logiciels [130] ont pour but de réduire la complexité des logiciels tout en améliorant la qualité. Ils offrent une meilleur structuration du code que dans le paradigme de programmation objet et par conséquent, ils permettent de mieux masquer l'implantation. Un composant est une « *boîte noire* » dont les interfaces sont clairement définies par un contrat. Ce contrat est le plus souvent exprimé dans un méta-langage tel que l'IDL. Les composants ont pour particularité de pouvoir être composés. On peut connecter des composants entre eux via leurs interfaces. Ces connexions sont limitées et définies par le contrat décrivant les interfaces. Ainsi un composant peut être connecté à n'importe quel autre composant remplissant le contrat. Cette propriété permet, a priori, une forte réutilisabilité et modularité des composants.

La figure 1.16 présente la représentation usuelle d'un composant CCM (CORBA Component Model) [106]. Ces composants ont différents types d'interfaces (ou points de connexion) : les facettes, les réceptacles, les sources et les puits d'évènements. Ces interfaces vont par paires : les facettes fournissent des services que les réceptacles utilisent et les sources émettent des évè-

nements que les puits consomment. En plus de ces interfaces, nous avons quelques interfaces particulières comme les interfaces composants, qui sont un point d'entrée permettant l'introspection des composants, et les attributs permettant de configurer les composants.

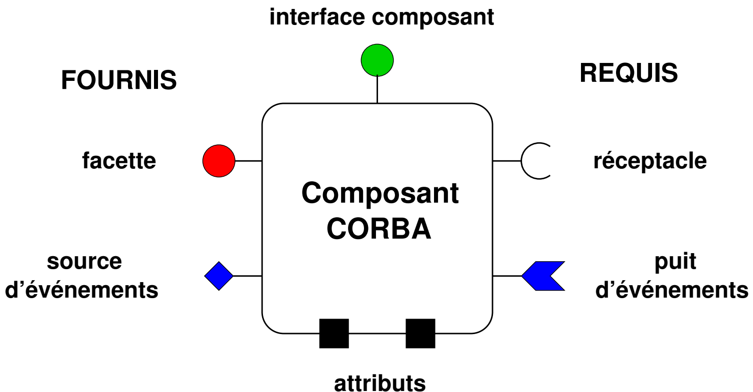


FIGURE 1.16 – Un composant CORBA (CCM)

Dans le contexte du couplage de codes, on peut utiliser des composants pour modéliser les codes, ou du moins s'en inspirer. Ainsi dans le contexte d'une plate-forme orientée composant, l'interface des codes couplés est spécifiée, ce qui permet de sous-traiter leur développement à des spécialistes du domaine d'application et de se concentrer uniquement sur la problématique du couplage.

Le modèle de composant le plus répandu dans le cadre de la simulation haute performance est sûrement CCA (Common Component Architecture) [6, 12]. Ce modèle composant à été défini par le CCA Forum [47] composé d'un ensemble de laboratoires et d'universités américaines. Contrairement aux composants CCM, les composants CCA n'ont que des facettes et des réceptacles décris grâce au langage SIDL (Scientific Interface Description Language). De plus, ces composants peuvent être parallèles selon un modèle SPMD. On peut également spécifier comment les composants sont connectés. Pour ce faire nous avons deux manières possibles, soit par des connexions directes, soit par des connexions réseaux. Les connexions directes permettent de spécifier que les composants sont dans le même espace mémoire. Par conséquent, des composants connectés directement peuvent accéder facilement aux zones mémoire les uns des autres. Les connexions réseaux sont des connexions distantes. Ces connexions ne sont pas basées sur une technologie spécifique, elles sont à la charge de l'implantation de la spécification CCA. Les implantations les plus courantes de CCA sont Ccafeine [2] (voir figure 1.17), XCat [79] et SCIRun2 [145]. Ce sont respectivement des implantations mémoire partagée, distribuée via des *web-services*, et basées sur MPI.

Maintenant que nous avons présenté ce que sont les composants logiciels et plus particulièrement les composants CCA, nous allons présenter quelques uns des environnements de couplage existants. Ces environnements modélisent tous les codes afin de pouvoir les coupler plus facilement. Nous pouvons distinguer plusieurs catégories. Nous avons tout d'abord les PSE [50] (Problem Solving Environment) qui sont des environnements tout intégrés et qui offrent des outils pour résoudre une classe de problème bien particulier. Il existe également des environnements de couplage plus génériques tels que Prism [135] et Kepler [3, 76]. Nous allons maintenant présenter certains de ces environnements plus en détails.

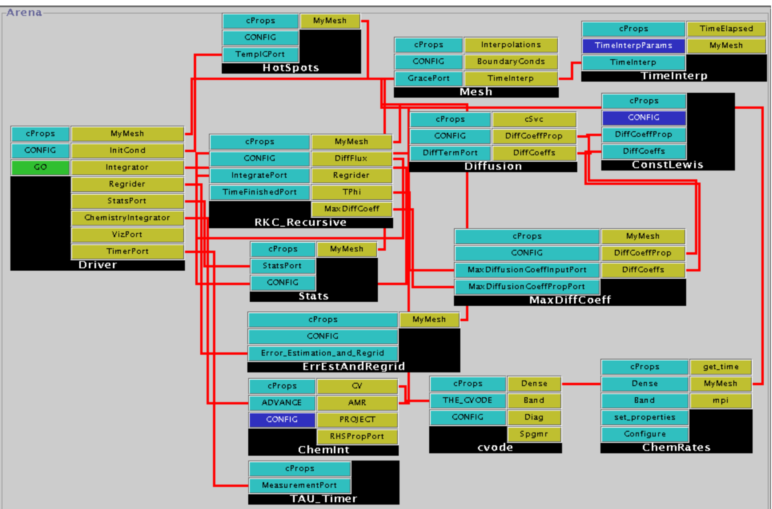


Figure 1.17 – Exemple de simulation décrite dans Ccafeine

SciRun2 [145] est un PSE basé sur SciRun [111] et CCA permettant de construire et de piloter des simulations numériques. SciRun2 est en fait une implantation de CCA permettant de connecter entre eux des composants CCA et CCM. Par conséquent, on peut construire des simulations à partir de tous les composants déjà existants dans d'autres environnements. SciRun2 permet de définir des composants distribués et également parallèles. Les échanges parallèles de données entre les composants sont pris en charge par des PRMI (Parallel Remote Methode Invocation) [29] qui sont une implantation de la spécification CCA M×N [13]. Dans SciRun2, les simulations sont donc modélisées comme un *network* qui décrit une composition de *modules*. Ces *modules* sont des composants logiciels contenant les codes de simulation spécifiques.

Cactus [19, 31, 53] est un PSE orienté *Grid Computing*. Il permet de construire des simulations parallèles et distribuées en englobant les codes de simulation dans des composants logiciels appelés *épines* (ou *thorns*). Ces composants sont combinés entre eux en les connectant à la *chair* (ou *flesh*). Les *thorns* sont décrites à l'aide d'un méta-langage, le CCL (Cactus Configuration Language). Afin de pleinement définir une *thorn*, il faut décrire les paramètres d'entrées, les données et fonctions accessibles par les autres *thorns* ainsi les fonctions que la *flesh* doit exécuter. Les données sont limitées à des tableaux multi-dimensionnels. Les *thorns* peuvent être parallèles et utiliser des bibliothèques de communication telles que MPI ou PVM. De plus, Cactus fournit des *thorns* prédéfinies permettant entre autre tout ce qui est gestion des données. Ainsi une *thorn* nommée *CactusPUGH* à pour rôle de gérer les redistributions et interpolations des grilles uniformes de données entre les *thorns* de calcul. *CactusPUGH* s'appuie sur MPI pour les communications parallèles. D'autre *thorns* servent à gérer les conditions aux limites, ou bien encore des moyens de résoudre des systèmes d'EDP, ou d'effectuer des I/O.

Palm [115] est un coupleur dynamique de codes. Il décrit les codes comme un ensemble de fonctions qui peuvent être séquentielles ou parallèles. Ces fonctions sont nommées des *Unit*. Ces *Unit* peuvent être agencées suivant des séquences, des boucles conditionnelles, *etc.*. Ces agencements dans Palm sont des *Branch* (voir figure 1.18). De plus, on peut avoir différents niveaux de parallélisme, soit du parallélisme de tâches, avec des *Branch* s'exécutant en parallèle, soit du parallélisme dans les *Unit*. Les connexions entre les *Unit* sont en fait des communications et les

unités de calcul peuvent produire des données (*PALM_Put*) ou en consommer (*PALM_Get*). Si les unités sont parallèles, PALM effectue les redistributions nécessaires en interne, les communications s'effectuant au-dessus de la bibliothèque MPI. Les *Branch* sont construites grâce à une interface graphique permettant de rajouter de façon dynamique des *Unit*.

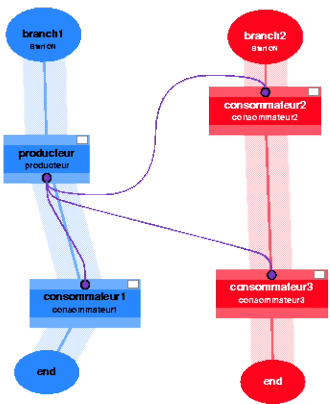


FIGURE 1.18 – Exemple de *Branch* Palm, avec une *Unit* de production de données et trois *Unit* de consommation

FCA (Flexible Coupling Approach) anciennement GCF (General Coupling Framework) [27, 46] est une approche permettant la composition et le déploiement d'applications couplées. Cette approche est basée sur des *models* qui sont des unités de composition. Chaque *model* correspond à une unité de développement, c'est-à-dire au code correspondant à un modèle mathématique ou physique. Chaque *model* a une interface avec le monde extérieur, des entrées et des sorties. Ainsi pour définir un *coupled model*, il faut décrire les *models* avec des méta-données selon le DCD (model Description, Composition, Deployment). La description contient des informations de haut niveau sur les données. Par la suite, la composition des *models* se fait via un paradigme de get/put. Cette approche est donc assez similaire à une représentation en composants. Il existe une implantation de cette approche, nommée BFG (Bespoke Framework Generator) [14], basée sur une description XML des méta-données, et des communications MPI pour les get/put natifs. Il est également possible de transférer les données via le coupleur Prism [118]. Les données supportées sont uniquement des tableaux ou des scalaires.

PCI (Potential Coupling Interface) [16, 17, 18] est une approche décrivant le code ainsi que les points de couplage potentiels en s'appuyant sur un *control-flow graph* (CFG) appelé PCI. La description des couplages entre deux PCIs est ensuite faite par le programmeur à l'aide d'un méta-langage, le CDL (Coupling Description Language). À chaque point de couplage (*coupling points*) sont associées des variables pouvant être importées ou exportées. Une fois les *coupling points* décrits dans chaque code de façon indépendante, nous pouvons les relier ensemble. Pour ce faire, il faut associer des actions aux points, ces actions pouvant être de trois types : *Send*, *Update* ou *Store*. Les actions *send* et *store* permettent les échanges de données entre les codes. Ces échanges se font en *TCP/IP*. La figure 1.19 montre un exemple de couplage de deux codes avec l'environnement PCI.

ESMF (Earth System Modeling Framework) [39, 63] est une plate-forme visant à standardiser la modélisation des systèmes planétaires. C'est un environnement orienté composant, fortement

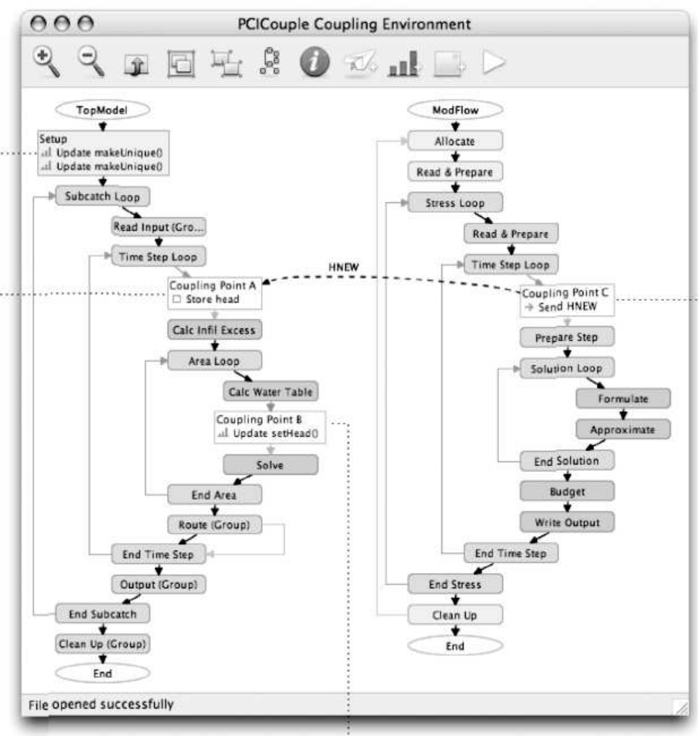


FIGURE 1.19 – Exemple de couplage de codes avec l'environnement PCI

inspiré de CCA. ESMF a une architecture en « *sandwich* », c'est-à-dire que le code de simulation est considéré comme un composant utilisateur qui est inséré entre deux autres couches, une couche de super-structure et une couche d'infrastructure. La couche de super-structure a pour rôle d'extraire les entrées/sorties du code pour pouvoir les connecter à un composant. Cette couche contient les bases de description et de composition des composants dans ESMF, les composants de grille, les composants de couplage et surtout les états ESMF (*ESMF states*). Les états ESMF sont en fait la description des données que l'on peut soit exporter, soit importer dans un composant. En particulier, le composant de couplage a pour rôle de récupérer les états exportés par un composant pour les importer dans un autre composant. La couche d'infrastructure, quant à elle, contient des outils de bas-niveau aidant au développement des simulations, c'est-à-dire les tableaux de données ainsi que leurs méta-informations sur les données physiques contenues et leurs distributions entre les codes. La distribution sur les codes est gérée via les composants de décomposition de domaine (*Decomposition Element Layout* ou *DELayout*). Ce sont ces distributions qui servent aux composants de couplage pour déterminer les redistributions à faire. Le composant *DELayout* contient également les routines de communication basées initialement sur la bibliothèque MPI et sur de la mémoire partagée dans le cas de *threads*.

Tous ces environnements permettent donc de définir des couplages de codes de haut niveau. Ils permettent de définir des couplages plus génériques que les couplages « *ad-hoc* » faits à partir des environnements de redistribution ou de communication. Cela est possible par l'introduction d'une modélisation de haut niveau des simulations. Ainsi, on peut noter que la représentation des simulations permet d'avoir une abstraction suffisante pour pouvoir les coupler et ce sans connaissance autre que leurs modélisations. Deux codes ayant la même modélisation peuvent donc être inter-changés dans un couplage. D'autre part, les environnements de couplage sont les

seuls à donner une représentation d'assez haut niveau des simulations pour pouvoir offrir des fonctionnalités de pilotage.

Nous allons à présent décrire plus en détails les environnements de pilotage. Certains de ces environnements sont aussi des environnements de couplage que nous avons déjà présentés.

1.4 État de l'art sur le pilotage de simulations

Nous avons présenté une vue d'ensemble des simulations numériques parallèles distribuées et des environnements de couplage pour ces simulations. Nous allons maintenant présenter le pilotage de simulations ainsi que les environnements existants permettant de piloter des simulations.

1.4.1 Introduction au pilotage de simulations

Le pilotage de simulations est une discipline dont l'utilité n'est plus à prouver. Depuis près de vingt ans, les scientifiques ont reconnu l'utilité de la visualisation scientifique tout d'abord pour l'analyse post-mortem des résultats produits par la simulation, mais aussi au cours même de l'exécution de ces simulations. Ainsi déjà dans un rapport de l'*US National Science Foundation* de 1987 [87], le pilotage de simulations numériques est défini comme un outil utile pour le procédé de recherche scientifique. Ce constat, bien que vieux de plus de 20 ans, reste encore vrai de nos jours. Ainsi le *pilotage de simulations* ou *computational steering* permet de visualiser en « *en-ligne* » les modifications engendrées par le changement d'un paramètre de la simulation, ce qui permet de mieux comprendre les relations de causes à effets, et de retrouver ainsi une démarche expérimentale. Les bénéfices d'une telle méthode ont été démontrés par *Marshall* et *al.* dans un article précurseur [86].

> « *A significant increase in productivity and comprehension is shown when steering is used.* »

Dans [86], les auteurs décrivent une solution spécifique pour le pilotage et la visualisation d'un modèle de turbulence 3D du lac Erie et ils mettent en avant le fait que le pilotage a permis de diminuer le temps entre le changement d'un paramètre et la visualisation des résultats. Il en résulte ainsi une amélioration de la productivité globale.

Dans la littérature, on trouve beaucoup de définitions du pilotage de simulations. Ainsi *Mulder* et *al.*, dans [98], définissent le pilotage de simulations comme étant le contrôle interactif du processus de calcul en cours d'exécution. Pour *Vetter* et *al.* [137], le pilotage de simulations est le contrôle d'une application et de ses ressources en cours d'exécution dans le but « *d'expérimenter* » les paramètres de l'application ou d'en améliorer les performances. Pour *Parker* et *al.* [112], le pilotage de simulations est un moyen d'extraire de manière efficace des données scientifiques d'une simulation numérique, mais également de les modifier de façon cohérente. Pour *Brooke* et *al.* [15], le pilotage de simulations est un procédé permettant de fournir aux physiciens un moyen d'interagir avec la simulation pendant qu'elle s'exécute, par exemple pour suivre l'évolution de certains paramètres, voire de les modifier. Et pour finir *Esnard* [40] présente le pilotage de simulations comme étant un cas particulier du couplage de codes, entre un code de simulation et un code de visualisation, permettant d'échanger les données de façon cohérente d'un code à l'autre.

Au vue de ces définitions, on peut décrire le pilotage comme un moyen d'interagir avec les données d'une simulation numérique, de façon cohérente et ce au cours même de l'exécution de la simulation afin de permettre aux scientifiques de mieux interpréter l'influence de certains paramètres (analyse de la dynamique des phénomènes, étude de sensibilité).

On trouve beaucoup de solutions « *ad-hoc* » comme celle du pilotage du modèle de turbulence 3D du lac Erie [86]. Mais ces solutions n'étant pas très viables en dehors du cadre dans lequel elles ont été développées, il apparait nécessaire de définir des environnements de pilotage dont le but est d'offrir une solution plus générique et réutilisable. Ainsi depuis 20 ans, on a vu apparaître beaucoup d'environnements de pilotage. Parmi les plus cités, on trouve VASE [58], Progress [136], Magellan [138], VIPER [121], CSE [97], SCIRun [111] ou CUMULVS [51], même si la recherche concernant la plupart de ces environnements ne semble plus active. On peut globalement tous les décrire au moyen du schéma 1.20.

FIGURE 1.20 – Architecture d'un environnement de pilotage.

Sur cette figure 1.20, on peut noter qu'un environnement de pilotage peut être divisé en trois parties : la *simulation numérique*, l'*interface utilisateur* et un *système de communication* entre les deux. L'interface utilisateur permet donc à l'utilisateur d'interagir de deux manières avec la simulation. Il peut soit la piloter (*steering*), soit visualiser ses données (*monitoring*).

Steering – D'une manière générale le *steering* consiste à modifier le comportement de la simulation. Cela peut se faire par la modification de paramètres du code, mais cela peut également permettre de contrôler le flot d'exécution, par exemple pour mettre la simulation en pause.

Monitoring – À l'inverse, le *monitoring* consiste à observer le comportement de la simulation. Pour cela on va extraire les données de la simulation au cours de son exécution. Ces informations peuvent être de différents types, comme des paramètres dont on veut suivre l'évolution, ou bien des informations sur l'évolution du flot d'exécution. Mais cela peut également être des données plus complexes permettant de visualiser les structures de données internes au code via des procédés de visualisation scientifique.

1.4.2 Environnements de pilotage existants

Nous allons présenter quelques uns des environnements de pilotage utilisés dans les simulations numériques actuelles. Pour ce faire, nous allons tout d'abord préciser les différents critères que nous allons décrire pour chacun de ces environnements.

1.4.2.1 Caractéristiques des environnements de pilotage

Nous allons diviser les caractéristiques en trois parties : les simulations numériques visées, l'architecture de l'environnement et l'interface graphique.

Simulations numérique visées – Tous les environnements de pilotage ne permettent pas de piloter tous les types de simulations. Historiquement, les environnements ne prenaient en compte que des simulations séquentielles ou parallèles en mémoire partagée. De nos jours, les simulations visées sont typiquement des simulations parallèles de types SPMD et des simulations distribuées. Parmi ces dernières, il y a plusieurs cas, des simulations dont les éléments sont distribués, mais également des simulations couplées.

Nous allons également voir si l'environnement vise des simulations déjà existantes, ou s'il faut construire la simulation dans l'environnement pour la piloter.

Pour finir, nous allons définir comment la simulation est modélisée par l'environnement afin d'avoir une vue plus générique. Ce critère n'est pas applicable pour tous les environnements car pour certains, il n'y a aucune abstraction faite de la simulation. Dans ce cas, nous verrons tout de même comment faire pour pouvoir piloter la simulation.

Architecture – Nous allons également détailler le type d'architecture de chaque environnement. Nous en distinguerons trois : les PSEs (Problem Solving Environments), les architectures client/serveur, et enfin les architectures client/serveur/client (voir figure 1.21).

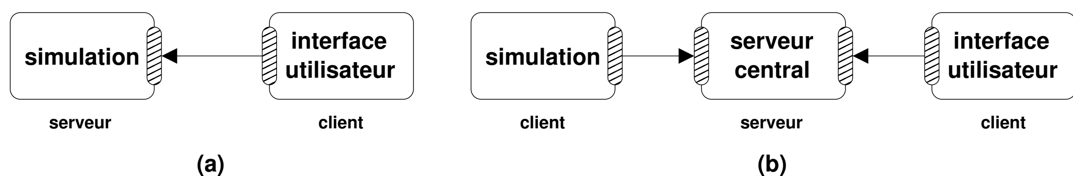


FIGURE 1.21 – Architecture client/serveur (a) et client/serveur/client (b)

Dans le cas client/serveur (a), la simulation joue la plupart du temps le rôle de serveur, et permet la connexion d'un ou plusieurs clients de pilotage. Les clients une fois connectés peuvent envoyer des requêtes de pilotage ou de monitoring à la simulation. Pour les architectures de type client/serveur/client (b), la partie algorithmique de pilotage et la partie visualisation sont séparées. La simulation se connecte à un serveur central gérant le pilotage et l'interface utilisateur en fait de même. Les requêtes venant de l'interface utilisateur sont donc interprétées par le serveur central avant d'être envoyées à la simulation.

Dans tous les cas, le pilotage et le monitoring sont dirigés par des requêtes venant de l'utilisateur. Ces requêtes peuvent être gérées de différentes manières. Les requêtes de pilotage doivent être exécutées de façon cohérente dans le temps. Dans le cas de simulations parallèles, cela veut dire qu'elles sont exécutées au même *moment*, c'est-à-dire à la même itération, ce qui correspond au même temps physique simulé sur tous les processus. Cela veut donc dire qu'il faut un moyen de synchroniser le traitement des requêtes de pilotage au niveau de la simulation. Pour cela, il existe plusieurs solutions, la synchronisation forte (*i.e.* une barrière MPI) ou la synchronisation faible (*loose synchronization* [51]). Dans le cas de la synchronisation forte, tous les processus sont stoppés au même moment, généralement sur un point d'instrumentation avant de traiter la requête de pilotage, le point d'instrumentation étant du code propre à l'environnement de pilotage ajouté dans le code source de la simulation. Ainsi si l'environnement est basé sur MPI,

une synchronisation forte pourra être obtenue par l'ajout d'un appel à la fonction `MPI_Barrier`. Dans le cas de la synchronisation faible, au lieu de stopper les processus, on prévoit le moment auquel les traitements devront être effectués.

Notons que l'on retrouve la même problématique de synchronisation pour les requêtes de monitoring car il faut garantir la cohérence des données extraites avant de les envoyer vers l'interface utilisateur. Par ailleurs, les environnements se distinguent par le système de communication qu'ils utilisent. Il y a plusieurs modes de communication, par passage de messages, par appel de méthode à distances ou bien encore via un modèle de plus haut niveau de type *data-flow*. Dans le cas du modèle *data-flow* les données sont échangées entre des modules (la simulation, ou une partie de la simulation, un client de pilotage) d'après un modèle de producteur/consommateur.

Interface utilisateur – Pour finir, nous avons les différentes interfaces graphiques possibles. Ces interfaces peuvent être une simple API permettant aux utilisateurs de définir leurs propres interfaces de visualisation à l'aide de bibliothèques graphiques telles que QT [119] ou OpenGL [108], ou bien des bibliothèques de visualisation scientifique telles que VTK [141]. Mais elles peuvent également être de plus haut niveau tels que des logiciels de visualisation scientifique comme AVS [7] ou Paraview [110].

1.4.2.2 Environnements de pilotage existants

Maintenant que nous avons défini les critères de classification des environnements de pilotage, nous allons étudier les environnements existants les plus représentatifs de l'état de l'art. Nous allons présenter plus en détails les environnements suivants : SCIRun, Cactus, RealityGrid, VISIT, CUMULVS, Discover et EPSN.

SCIRun [111, 113, 142], comme nous l'avons déjà vu précédemment (section 1.3.3), est un PSE (Problem Solving Environment). SCIRun intègre du pilotage de simulation alors que SCIRun2 est encore à l'état de prototype. SCIRun permet de construire des simulations en connectant des *modules* via des *datapipes*. Les *datapipes* sont des flux de données reliant plusieurs *modules*. Une simulation construite de la sorte est appelée un réseau (*network*). Ainsi SCIRun permet de construire des simulations selon un paradigme *data flow*. SCIRun est utilisé dans le cadre de la recherche biomédicale ainsi que pour l'étude de la propagation du feu. Par ailleurs, SCIRun gère le parallélisme de tâches en permettant l'exécution de plusieurs modules en même temps, mais il permet également d'intégrer des *modules* multi-threads. Nous avons vu que SCIRun2 permettait d'avoir des *modules* parallèles pas uniquement limités au paradigme de mémoire partagée, mais ce n'est encore qu'un prototype qui n'est pas intégré pour la visualisation scientifique. De plus, dans la version actuelle de SCIRun, il est possible d'intégrer des simulations existantes en découpant les différentes tâches de cette simulation en modules SCIRun. Là encore SCIRun2 offre plus de liberté, car il permettra l'intégration de composants parallèles.

Afin de pouvoir piloter les simulations, SCIRun offre la possibilité de définir des *paramètres* permettant de configurer les *modules*. Ces paramètres sont modifiables en cours d'exécution. SCIRun distingue deux types de modifications pour ces paramètres, synchrone ou asynchrone. Dans le cas synchrone, la modification est prise en compte « *à la volée* », c'est-à-dire en cours d'exécution du *module* si ce dernier le permet par exemple s'il est sans état. Dans le cas où le *module* ne permet pas une telle modification, il est arrêté et réexécuté. Ce type de modification n'est possible que lorsque le module n'a pas d'état, ou lorsque la modification est limitée à un

seul *module*. Dans le cas asynchrone, la modification est prise en compte à la prochaine exécution du *module*, c'est-à-dire à l'itération suivante et on parlera de *feedback loop*.

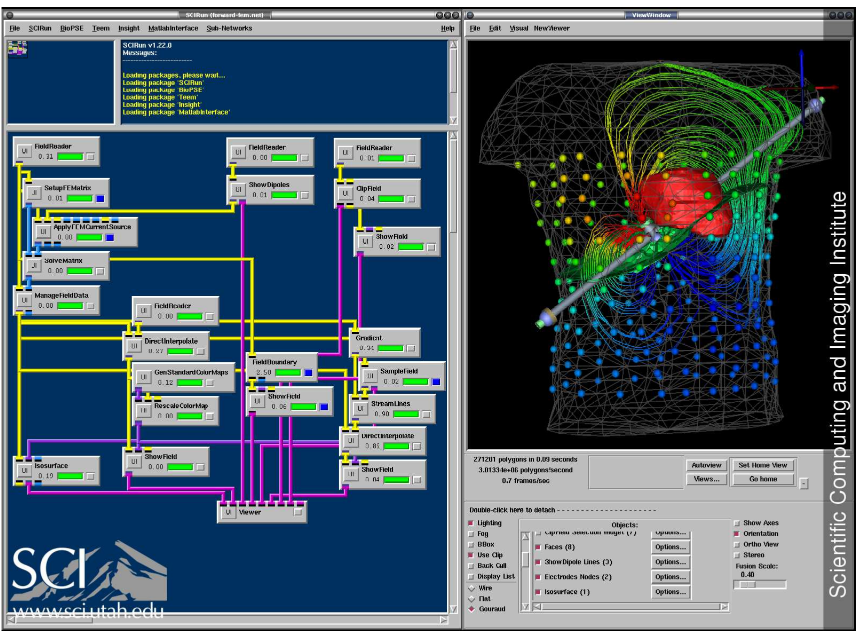


FIGURE 1.22 – Simulation de défibrillateur dans SCIRun

SCIRun permet en plus de piloter les simulations, de les surveiller (*monitoring*). La visualisation scientifique passe par des *modules* de traitement de données et ce jusqu'à la phase de rendu dans un *module* d'affichage. Certains de ces *modules* permettent également par le biais de la *feedback loop* de modifier visuellement les paramètres des *modules*. La figure 1.22 présente un exemple de l'environnement SCIRun sur une simulation de défibrillateur. Sur la gauche on peut voir le *network* et un module de visualisation sur la droite.

Cactus [53] est un PSE permettant la conception de simulations parallèles. Comme nous l'avons vu précédemment (section 1.3.3), Cactus est basé sur une vision composant des simulations. Les codes sont dans des *thorns* interconnectées entre elles par la *flesh*. Il est possible d'intégrer des simulations existantes en les transformant en *thorns*, même si comme dans tous les PSE, il est plus simple de construire une simulation propre à l'environnement. Historiquement Cactus a été développé pour l'étude des trous noirs par la simulation numérique.

Afin de pouvoir piloter ces simulations, certains paramètres des *thorns* doivent être déclarés comme pilotables (*steerable*). Quand le développeur d'un *thorn* décrit un tel paramètre, il assure que les modifications de ce paramètre entraîneront une modification sur la simulation. Une fois ces paramètres décrits, ils peuvent être modifiés en cours d'exécution par les outils inclus dans Cactus. Le *monitoring* des simulations se fait quant à lui via des *thorns* particulières, capables de récupérer les données et de les exporter vers des logiciels extérieurs, la méthode la plus utilisée étant l'extraction au format HDF5 permettant d'écrire les données dans des fichiers ou de les *streamer* via le réseau. De plus, les données peuvent être dégradées afin d'augmenter la vitesse de transfert de ces dernières.

Cactus permet de visualiser ou de piloter les paramètres d'une simulation via plusieurs types d'interfaces. On peut piloter les paramètres via des interfaces légères telles qu'un portail web, mais on peut également utiliser n'importe quel logiciel permettant de lire des fichier HDF5. De

plus, Cactus permet de transférer des isosurfaces, pré-calculées par une *thorn* via un protocole propriétaire, vers l'outil de visualisation Amira [4].

RealityGrid [15, 26, 116] permet via la *RealityGrid Steering Library* de piloter des simulations parallèles et couplées existantes. Le pilotage se fait par une instrumentation du code source de la simulation auquel il faut ajouter des appels à la bibliothèque. Ces appels servent à initialiser l'environnement de pilotage et également à enregistrer les variables pilotables ou « *monitorables* ». Enfin des appels à la bibliothèque de pilotage placés dans la boucle principale de la simulation permettent à RealityGrid de vérifier s'il y a des requêtes venant d'un client.

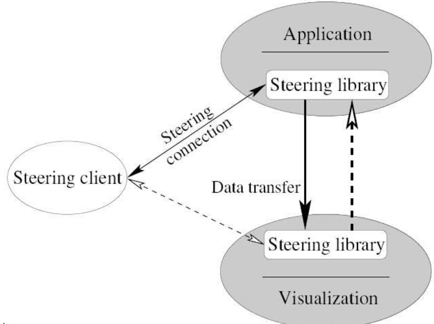


FIGURE 1.23 – Architecture de RealityGrid pour le pilotage de simulations

Le pilotage de simulations dans RealityGrid est basé sur une architecture client/serveur, ou dans le cas de la version orientée grille, sur une architecture client/serveur/client. La simulation joue le rôle de serveur après avoir été instrumentée grâce à l'API de la bibliothèque. La bibliothèque de pilotage permet de traiter les demandes venant des clients. Le client de pilotage se connecte à la partie de la *steering library* présente dans la simulations (voir figure 1.23). Dans le cas du Grid Computing, l'environnement RealityGrid met en place un *Steering Grid Service* (SGS). Le client se connecte donc au SGS qui lui même se connecte à la simulation. Les actions de pilotage possibles sont *pause*, *resume*, *detach*, *stop* et *rewind*. Le client envoie une requête au SGS qui la renvoie à la simulation. La simulation l'exécute quand elle arrive sur un point d'instrumentation. Le client peut également demander une reprise sur *checkpoint* si la simulation le permet. On peut donc faire un *rewind*, ou un retour sur un *checkpoint* bien précis. Pour les données, la simulation enregistre des données pilotables (accessibles en écriture uniquement), ou « *monitorables* » (en lecture seulement). RealityGrid permet de récupérer les données vers un client de visualisation. Pour ce faire, il faut que le client ait été modifié à l'aide de la bibliothèque de pilotage. Dans ce cas, les transferts de données se font soit via des fichiers XML, soit via le protocole SOAP. RealityGrid permet également le pilotage de simulations couplées en mettant en place une hiérarchie de SGS. Chaque code à un SGS associé, puis les SGS sont connectés à un SGS parent. Dans ce cas, le client se connecte au SGS le plus haut dans la hiérarchie.

RealityGrid met à disposition une interface graphique générique en QT [119] permettant de se connecter à n'importe quelle simulation préalablement instrumentée afin de pouvoir la piloter et de visualiser les données produites.

VISIT (VISualisation Interface Toolkit) [36, 124, 139] est une bibliothèque permettant de connecter une simulation à un outil de visualisation. VISIT permet de visualiser et de modifier les données de simulations parallèles existantes. Pour ce faire, il faut ajouter des appels à l'API

« *client* » de VISIT dans la simulation. Des projets tels que le DMMD (Distributed Memory Molecular Dynamics) [33] pour des simulations de dynamique moléculaire utilise l'environnement VISIT.

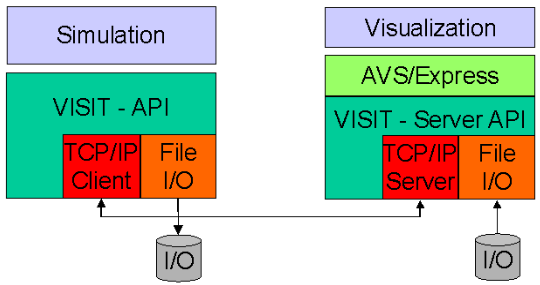


FIGURE 1.24 – Architecture client/serveur de VISIT

VISIT est basé sur une architecture client/serveur (voir figure 1.24), la simulation jouant le rôle de client et l'outil de visualisation étant le serveur. Ainsi toutes les décisions sont à l'initiative de la simulation. C'est la simulation qui envoie des données au serveur de visualisation, c'est également elle qui récupère les données du serveur afin de modifier les paramètres. Les transferts de données se font via le protocole TCP/IP et utilisent des messages « *à la MPI* », c'est à dire que les messages contiennent un tag spécifiant le type de données. VISIT permet également de stocker les données de la simulation sur disque afin de les « *rejouer* » ultérieurement dans le serveur de visualisation.

VISIT implante des serveurs de visualisation pour Paraview [110] et AVS/Express [7]. De plus dans les dernières versions, VISIT propose un multiplexeur de serveurs pour pouvoir faire de la visualisation collaborative.

CUMULVS (Collaborative User Migration User Library for Visualization and Steering) [51, 78] est un environnement de pilotage basé sur PVM et développé par l'ORNL (Oak Ridge National Laboratory). CUMULVS permet de piloter des simulations parallèles existantes basées sur les bibliothèques PVM ou MPI. Pour ce faire, il faut instrumenter le code en rajoutant un point d'instrumentation dans la boucle principale du code.

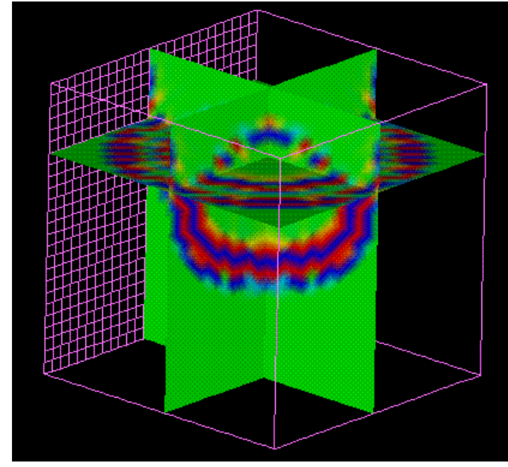


FIGURE 1.25 – Visualisation de la propagation d'une onde acoustique avec CUMULVS

CUMULVS est basé sur une architecture client/serveur, avec la simulation jouant le rôle de serveur et les interfaces de visualisation celui de client. Plusieurs clients peuvent se connecter et se déconnecter dynamiquement à une simulation. CUMULVS permet aux clients de récupérer des données de la simulation. Pour ce faire, la simulation décrit les données qu'elle veut exporter ainsi que leur stockage. Ces données peuvent être, soit des tableaux denses multi-dimensionnels distribués en blocs cycliques à la HPF [77], soit des particules. Lorsqu'un client de visualisation effectue une requête pour extraire des données, celle-ci est traitée au niveau de la simulation par un algorithme de *loose synchronization*. Cela consiste à prévoir l'itération à laquelle les données devront être envoyées. Cette itération est choisie par le client. Le client peut choisir la première itération, ainsi qu'une fréquence pour répéter la requête. Une fois cette itération atteinte par un processus, lors du passage sur le point d'instrumentation, les données sont bufferisées et envoyées de manière asynchrone durant l'itération. La simulation reste bloquée lors du passage suivant sur le point d'instrumentation jusqu'à ce que le client la débloque par l'envoi d'un ordre « *continue* ». Si cet ordre est envoyé assez tôt, la simulation ne se bloquera pas. Les clients peuvent aussi piloter des paramètres de la simulation. Pour cela, il faut déclarer les paramètres pilotables dans la simulation. Les clients peuvent modifier ces paramètres par un procédé analogue à la récupération de données. Pour gérer les clients multiples, CUMULVS utilise un système de jeton (*token*). Seul un client peut avoir le *token*. Ainsi, les clients doivent demander à obtenir le *token* avant de pouvoir piloter la simulation.

CUMULVS possède un client de visualisation en TCL/TK [132] et permet également de visualiser les données dans AVS (voire figure 1.25). Par ailleurs, un prototype de CUMULVS a été utilisé pour prototyper une implantation de la spécification CCA M×N. Il est également à noter que CUMULVS est toujours utilisé pour piloter des simulations telles que l'Ames Lab Classical Molecular Dynamics (ALCMD) [10, 11, 128], une simulation de dynamique moléculaire.

DISCOVER (Distributed Interactive Steering and Collaborative Visualization EnviRonment) [85, 99, 100] est un environnement à base de technologie web pour le pilotage de simulations numériques. Les simulations visées par DISCOVER sont des simulations parallèles et distribuées. Pour pouvoir les piloter, il faut transformer les différentes parties de la simulation en *objets interactifs*.

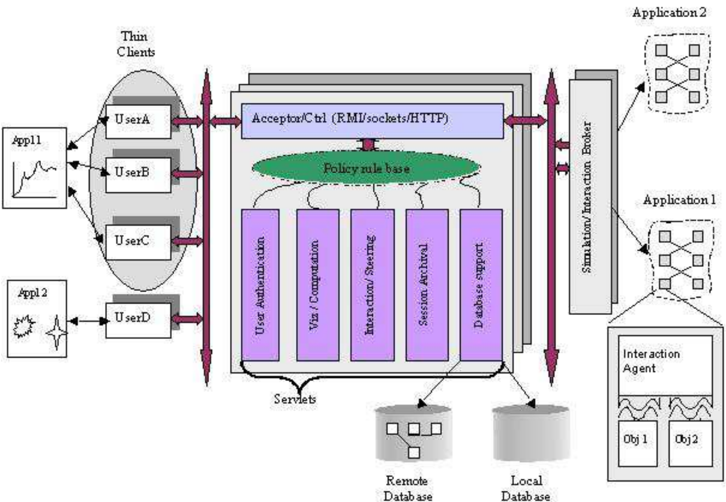


FIGURE 1.26 – Architecture de DISCOVER

DISCOVER est basé sur une architecture client/serveur/client (voir figure 1.26). La simulation est tout d'abord transformée en objets distribués à l'aide de DIOS (Distributed Interactive Object Substrate) [101], puis enrichie de *sensors* et *actuators* pour former des objets distribués dits interactifs. Les *sensors* et les *actuators* décrivent les données des simulations respectivement consultables et modifiables. De plus, ces objets disposent d'opérations *gather* et *scatter* afin de gérer les transferts de données vers/depuis un processus. Les *objets interactifs* peuvent exporter des *vues* et accepter des *commandes* en les enregistrant auprès des serveurs nommés *interaction servers* (IS). Ces IS gèrent tout ce qui est pilotage, collaboration, authentification des clients et sécurité. Les clients se connectent à ces IS pour former des groupes de collaboration. Les requêtes émanant des groupes sont traitées par les IS : elles sont ordonnancées et transmises aux objets distribués, puis les réponses sont envoyées en retour à tous les clients du groupe. Les requêtes de visualisation de données récupèrent les *Vues* correspondantes sur les objets distribués. Ces *Vues* sont des données texte, des courbes 1D, ou des iso-surfaces.

Les clients sont typiquement des clients web légers. En se connectant aux serveurs DISCOVER un client récupère via des applets Java le code nécessaire à l'interaction avec les simulations. Pour chaque type de *Vue*, le serveur a du code pré-enregistré qu'il envoie aux clients.

EPSN (Environnement pour le Pilotage de Simulations Numériques) [25, 42, 43] est comme son nom l'indique, un environnement de pilotage de simulations numériques. Il permet de piloter des simulations numériques parallèles existantes. On trouve des exemples d'utilisation de cet environnement sur des simulations de mécanique des fluides telle que FluidBox [45], ou de cosmologie telle que Gadget-2 [49]. Pour ce faire, il nécessite d'instrumenter le code des simulations par l'ajout de point d'instrumentation avant et après les tâches du code que l'on souhaite piloter. Ces tâches sont imbriquées de façon hiérarchique les unes dans les autres et il en découle un arbre de tâches.

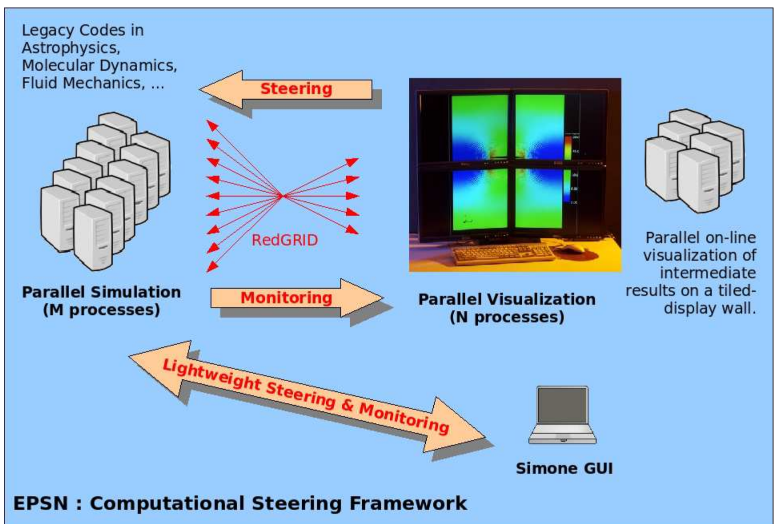


Figure 1.27 – Vue générale de EPSN

EPSN est basé sur une architecture client/serveur, la simulation jouant le rôle de serveur. Elle accepte la connexion et déconnexion dynamique de plusieurs clients de pilotage en même temps. Pour piloter les simulations, il faut décrire la structure du code sous forme de tâches hiérarchiques (tâches composées, en boucle ou conditionnelles). Cette description sert dans les algorithmes de planification de EPSN, afin de prévoir les dates auxquelles les requêtes seront exécutées. Ainsi,

EPSN est basé sur un principe de *loose synchronisation* comparable à CUMULVS. Les requêtes peuvent être de trois types : contrôle, interaction ou données. Les requêtes de contrôle permettent de mettre la simulation en pause, les requêtes d'interaction permettent l'exécution de méthodes pré-enregistrées dans le code (*callback*) et les requêtes de données permettent de consulter ou de modifier les données du code. Pour accéder aux données, il faut préalablement les décrire dans la simulation à l'aide de la bibliothèque RedGRID (voir section 1.3.2). Les données sont ensuite récupérées de façon asynchrone sans que la simulation soit, a priori, bloquée.

EPSN fournit un client de pilotage générique en QT [119], nommé Simone (SImulation MONitor for EPSN) [38], permettant de se connecter à n'importe quelle simulation (voir figure 1.27). Ce client permet via des plugins VTK (Visualization ToolKit) [141] de visualiser de façon simple les données. EPSN met également à disposition des plugins Paraview [110] afin de pouvoir facilement définir un client parallèle de visualisation. EPSN permet également via une API de développer des clients plus spécifiques.

1.4.2.3 Discussion

Nous avons résumé tous ces environnements dans le tableau récapitulatif 1.1. On note que tous ces environnements permettent le pilotage de simulations parallèles, même si pour la version actuelle de SCIRun, cela se limite à des simulations *multi-threads*. De plus, ces simulations sont en grande partie des simulations existantes, à l'exception des simulations visée par SCIRun dont la stratégie consiste à construire les simulations dans l'environnement. Il est à noter tout de même que dans SCIRun2 ces limitations devraient disparaître. Certains de ces environnements tels que RealityGrid, Cactus et SCIRun permettent de piloter des simulations couplées.

Pour ce qui est de la modélisation faite des simulations, les PSE, comme pour SCIRun ou Cactus, utilisent un découpage de la simulation en modules englobant plus ou moins de tâches de la simulation, cette modélisation n'étant pas faite dans un premier lieu pour le pilotage mais pour pouvoir ordonnancer et connecter les différents modules. Le plus souvent, la modélisation se limite à un simple point d'instrumentation dans la boucle principale de calcul (CUMULVS, RealityGrid). Seul EPSN fournit une modélisation de haut niveau des simulations dans le cadre du pilotage et ce à base d'un arbre de tâches hiérarchiques.

En ce qui concerne l'architecture de ces environnements, nous avons essentiellement des architectures client/serveur, client/serveur/client et des PSE. Seul un environnement se distingue, VISIT, avec une architecture client/serveur inversée, c'est-à-dire avec la simulation qui joue le rôle de client. Les architectures client/serveur/client sont des environnements qui sont orientés grille et cela permet de plus facilement gérer le problème des passerelles d'accès.

Pour traiter les requêtes, la plupart des environnements utilisent des procédés de synchronisation forte afin d'assurer la cohérence des données et communiquent ces données par passage de messages (RealityGrid, VISIT). Seuls CUMULVS et EPSN proposent un mécanisme de synchronisation faible qui a l'avantage d'être moins intrusif sur l'exécution des simulations. Les PSE peuvent être mis à part car la cohérence des traitements et les communications sont assurées de façon transparente par le modèle *data-flow*.

1.4.3 Les limites des environnements existants

Dans cette thèse, nous nous intéressons au pilotage de simulations couplées. Or la plupart des environnements que nous avons étudiés ne permettent pas la pilotage de ce type de simulation. En effet, comme nous l'avons déjà souligné (section 1.4.2.3), seuls les environnements Cactus, RealityGrid et bientôt SCIRun2 peuvent piloter des simulations couplées. Cactus et SCIRun2

Environnement	**Simulations numériques**		**Architecture**			**Visualisation**
	Type	**Modélisation**	**Type**	**Pilotage**	**Communication**	
SCIRun	mémoire partagée	modules	PSE	modification de paramètres	data-flow	modules intégrés
Cactus	Parallèles Distribuées	modules	PSE	modification de paramètres	data-flow	Web, Amira HDF5
RealityGrid	Parallèles Distribuées (couplées)	boucle de calcul	C/S/C	synchronisation forte	XML par fichier ou SOAP	GUI Qt
VISIT	Parallèles	envoi depuis la simulation	S/C	synchronisation forte	passage de messages	AVS/Express, Paraview
CUMULVS	Parallèles	boucle de calcul	C/S	synchronisation faible	passage de messages	TCL/TK, AVS
DISCOVER	Parallèles Distribuées	objets distribués	C/S/C		gather/scatter	Web, applets java
EPSN	Parallèles	arbre de tâches hiérarchiques	C/S	synchronisation faible	passage de messages	GUI QT, VTK parallèle
EPSN2	Parallèles Distribuées (couplées)	arbres de tâches hiérarchiques	C/S	synchronisation faible	passage de messages	GUI QT, VTK parallèle

TABLE 1.1 – Synthèse des environnements de pilotage et positionnement de EPSN2

étant des PSE, seul RealityGrid est un environnement générique basé sur l'instrumentation du code source, car les PSE nécessitent de modifier de façon importante les simulations existantes pour les intégrer dans l'environnement, ce qui selon nous représente une limitation importante. Mais contrairement à RealityGrid, les PSE fournissent une certaine modélisation des simulations qui permet d'offrir aux utilisateurs des interactions de plus haut niveau.

Pour les autres environnements, la modélisation de la simulation ne permet pas d'évoluer facilement vers la modélisation de simulations couplées. CUMULVS, par exemple, a une vision du code en boucle en temps unique, ce qui s'adapte mal au cas des codes couplés M-SPMD. Dans le cas de DISCOVER, la vision en objet distribué permettrait, a priori, de modéliser des simulations couplées. Mais à notre connaissance aucun résultat n'est publié dans ce sens. Pour EPSN, le modèle en tâche hiérarchique a été mis en place pour piloter des simulations SPMD. La modélisation permet en particulier d'assurer la cohérence temporelle des traitements. Mais cette modélisation ne permet pas de représenter les « *liens* » existants entre les différents codes d'une simulation couplée, comme des appels de méthodes sur des serveurs.

Par ailleurs, il faut noter que la « *bonne* » modélisation d'une simulation n'est pas une condition suffisante pour assurer la cohérence des interactions. Si nous prenons l'exemple de DISCOVER, il peut modéliser des simulations distribuées, par conséquent il peut modéliser des simulations couplées. Par contre, il ne modélise pas le lien qu'il y a entre les codes d'un couplage. Par conséquent, le traitement des requêtes de pilotage est cohérent localement à un code, mais pas globalement à la simulation couplée. Ce genre de pilotage pourrait être fait par n'importe quels environnements multi-simulations. Mais l'intérêt reste cependant limité car on ne peut pas assurer la cohérence des traitements globalement.

Pour finir, si nous prenons le cas de RealityGrid, la modélisation du couplage de codes se fait par l'ajout d'informations permettant de lier les données des différents codes. Mais la modélisation des simulations ne reste qu'une modélisation simpliste en points d'instrumentation. Cette information supplémentaire permet dans le cas du *monitoring* de récupérer les données à visualiser à la même itération dans les différents codes. Cette modélisation se limite donc à des couplages de codes SPMD avec des données cohérentes entre elles à chaque itération. Par conséquent, RealityGrid ne peut pas, a priori, prendre en compte des simulations Client/Serveur ou bien des simulations M-SPMD dont les données ne sont pas cohérentes entre elles à toutes les itérations des codes couplés.

Ces différentes limitations et le fait que les codes couplés sont de plus en plus présents dans les simulations HPC ont motivé nos travaux sur la conception d'un environnement de pilotage de simulations couplées.

1.5 Positionnement

Pour conclure ce chapitre, nous allons présenter comment nos travaux sur EPSN2 se placent par rapport aux environnements de pilotage que nous avons décrit. EPSN2 a pour but de piloter des simulations couplées décrites au début de ce chapitre, à savoir des simulations de type M-SPMD et Client/Serveur telles que nous les avons décrites dans la section 1.1.2. Notre environnement s'appuie sur EPSN ; par conséquent, nous allons conserver toutes les propriétés de ce dernier, c'est-à-dire une modélisation de haut niveau des simulations, une architecture client/-serveur de l'environnement. Afin de conserver la propriété de faible intrusion dans la simulation, nous continuerons d'utiliser un mécanisme de *loose synchronization*. Les clients de visualisation restent également potentiellement parallèles afin d'améliorer les temps de post-traitement des données. Il y a donc deux problèmes essentiels à traiter : définir une modélisation des simulations

couplées en vue de pouvoir les piloter et maintenir la cohérence des différents traitements et ce par le biais d'une modification des algorithmes de *loose synchronisation*.

1.5.1 Modélisation des simulations couplées

Afin de modéliser des simulations M-SPMD et Client/Serveur nous avons opté pour le *Modèle Hiérarchique en Tâches* (MHT) introduit initialement dans EPSN. Ce modèle permet de décrire des simulations SPMD sous forme de tâches hiérarchiques (tâche, boucle, conditionnelle, *etc.*.). Cette représentation est basée sur le fait que tous les processus doivent exécuter exactement le même code et donc avoir exactement la même description en tâches hiérarchiques. Par conséquent, on ne pourra pas l'utiliser tel qu'il est défini pour des simulations M-SPMD ou Client/Serveur. Nous allons devoir étendre ce modèle afin de le rendre plus générique. Notre nouveau modèle permettra non seulement de représenter des simulations couplées, mais également de se repérer de façon précise dans le flot d'exécution. Cette dernière propriété permettra de planifier la date de traitement des requêtes de pilotage et donc de mettre en place un algorithme de *loose synchronization* efficace. Cet algorithme est une partie essentielle de l'environnement de couplage car il permet de garantir la cohérence des traitements des requêtes.

De plus, le modèle en tâches hiérarchiques définit des plages d'accessibilités pour les différents traitements possibles : accès aux données en lecture/écriture ou appels de fonctions prédéfinies dans le code (*callback*). Cela permet de ne pas effectuer les traitements sur un point d'instrumentation comme dans RealityGrid, mais de recouvrir ces traitements sur toute une plage d'accès. Dans le cas de requêtes de *monitoring*, cela permet d'accéder aux données de façon asynchrone tout en évitant de faire une copie dans un cache, comme dans CUMULVS. Cela réduit potentiellement la plage d'accès aux données, mais dans le cas où le volume de données est important, une copie se révèle souvent trop coûteuse.

1.5.2 Cohérence des traitements

Comme nous l'avons dit, le modèle que nous allons introduire ainsi que les algorithmes de planification vont nous permettre d'assurer la cohérence des différents traitements. Par conséquent, nous devons définir précisément ce que nous entendons par cohérence car elle peut être de plusieurs types selon les traitements que l'on effectue. Dans le cas de simulations parallèles, cela se résume à exécuter les traitements au même moment pour chaque processus. Cela veut dire à la même itération et dans la même tâche pour tous les processus. Dans le cas de simulations distribuées, la cohérence ne s'exprime plus forcément de façon aussi simple. Il faudra donc clairement définir la notion de *traitement cohérent* pour chaque type de requête de pilotage.

Par exemple, dans le cas de données à visualiser, il faudra s'assurer que nous les exportons des codes de simulation de façon cohérente entre les codes et que nous assurons cette même cohérence dans le client jusqu'au moment où ces données sont totalement traitées. Par ailleurs, nous devons assurer cette cohérence pour tous les types de client, qu'ils soient séquentiels ou parallèles.

Chapitre 2

Modèle pour un environnement de pilotage de simulations distribuées

Sommaire

2.1 Introduction **42**
2.2 Modèle abstrait de EPSN **42**
2.2.1 Modèle de description de simulations SPMD 42
2.2.1.1 Modèle hiérarchique en tâches (MHT) 43
2.2.1.2 Système de date 45
2.2.1.3 Modèle de description des données 46
2.2.1.4 Exemple 47
2.2.2 Modèle de pilotage 48
2.2.2.1 Pilotage par les requêtes 48
2.2.2.2 Algorithme de planification dans le cas SPMD 50
2.2.2.3 Exemple 51
2.2.3 Limitations du modèle pour les simulations distribuées 52
2.3 Modèle abstrait pour les simulations distribuées **53**
2.3.1 Modèle de description de simulations distribuées 53
2.3.1.1 Modèle hiérarchique en tâches distribué (MHTd) 53
2.3.1.2 Système de dates 58
2.3.1.3 Modèle de description des données 60
2.3.2 Modèle de pilotage 62
2.3.2.1 Pilotage par les requêtes 62
2.3.2.2 Les algorithmes de planification 64
2.3.2.3 Cohérence des données dans les clients de pilotage 68
2.4 Conclusion **70**

2.1 Introduction

Afin de pouvoir piloter des simulations M-SPMD et Client/Serveur telles que nous les avons présentées dans le chapitre précédent, il faut définir une abstraction des codes sous-jacents pour que notre approche soit la plus générique possible. Nous allons baser le pilotage des simulations distribuées sur une modélisation des codes permettant la prise de décision de pilotage intelligente à partir d'une information de haut-niveau. Pour cela, il nous faut introduire une modélisation la plus simple possible, mais également la plus complète possible en fonction des interactions que l'on veut avoir avec les simulations. Dans le chapitre précédent (Sec. 1.4.2), nous avons présenté différents modèles de représentation d'une simulation, que ce soit pour des environnements de couplage ou de pilotage. Cependant, nous avons vu que ces modèles avaient des limitations et ne convenaient donc pas pour le pilotage des simulations distribuées que nous visons dans cette thèse. Notre approche pour le pilotage de ces simulations s'appuie sur le modèle abstrait de EPSN. Il a été modifié afin de lever les limitations l'empêchant de représenter et de piloter des simulations distribuées.

Dans l'état de l'art, nous avons brièvement présenté le modèle abstrait de EPSN (Sec. 1.4.2.2). Nous allons à présent introduire le modèle abstrait de EPSN de façon plus détaillée, afin de pouvoir mieux présenter les modifications effectuées dans ce modèle pour prendre en compte les simulations distribuées.

2.2 Modèle abstrait de EPSN

Comme le montre la figure 2.1, le modèle abstrait d'EPSN s'appuie sur deux sous-modèles, un modèle de description et un modèle de pilotage. Ces deux modèles ont été mis au point pour pouvoir piloter de façon efficace et cohérente des simulations parallèles. Dans la suite de cette section, nous allons détailler ces deux sous-modèles.

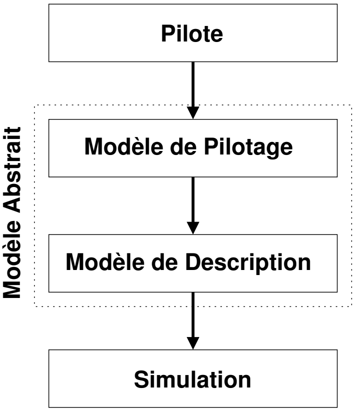


FIGURE 2.1 – Modèle abstrait de pilotage de EPSN

2.2.1 Modèle de description de simulations SPMD

Le modèle de description est découpé en trois parties : une modélisation du flot d'exécution, un système de date permettant de se repérer avec précision dans ce flot d'exécution et une modélisation des données utiles pour le pilotage. Ce modèle sert à décrire les éléments clés pour le pilotage des simulations.

2.2.1.1 Modèle hiérarchique en tâches (MHT)

Nous commençons par présenter le modèle de description du flot d'exécution utilisé dans la plate-forme EPSN pour des simulations parallèles et développé dans la thèse de *A. Esnard* [40]. Il définit un *Modèle Hiérarchique en Tâches* (MHT) inspiré d'un graphe de tâches hiérarchiques ou HTG (Hierarchical Task Graph) introduit par *Polychronopoulos* et *Girkar* [52] pour modéliser le flot d'exécution d'une application.

Le *MHT* a pour avantage d'être plus simple à utiliser dans notre contexte qu'un HTG. En effet le HTG est, par définition, un graphe orienté acyclique (ou DAG) alors que le *MHT* est basé sur un *arbre de tâches*. Cette représentation nous permet de capturer le flot d'exécution des programmes structurés propres aux langages impératifs de haut niveau (C, C++, Fortran, *etc.*). En contrepartie, le *MHT* ne peut pas représenter des ruptures dans le flot d'exécution d'un programme comme les instructions `goto` ou des exceptions.

Le *MHT* est donc basé sur un arbre de tâches. Ces tâches sont de quatre types de base (voir figure 2.2) : (a) les *tâches composées*, (b) les *tâches en boucle*, (c) les *tâches conditionnelles* et (d) les *tâches en point* qui sont des tâches dégénérées et qui ne servent que dans le cadre du pilotage. Toutes ces tâches sont délimitées par deux *points d'instrumentation* marquant le début et la fin de la tâche dans le code source. De plus, elles sont toutes *hiérarchiques* dans le sens où elles peuvent contenir d'autres tâches (hormis les tâches en point).

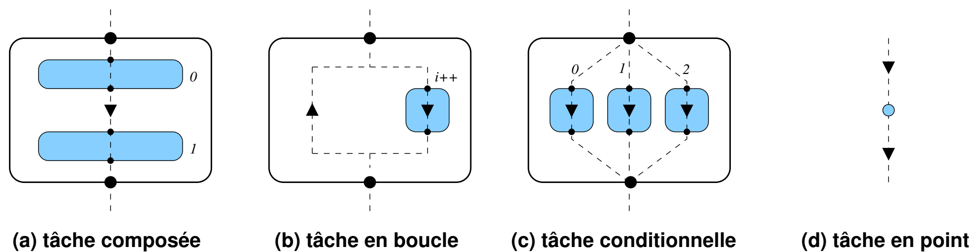


FIGURE 2.2 – Modèle abstrait de pilotage de EPSN

Nous allons décrire plus en détails ces différents types de tâches :

La *tâche composée* représente des blocs logiques de code ; lorsque nous avons aucune tâche imbriquée dans une *tâche composée*, nous parlerons de *tâche simple*. De plus, dans une tâche composée, nous allons numéroter les tâches d'après leur ordre dans la séquence.

La *tâche en boucle* capture les structures itératives d'un code. Ces tâches contiennent une seule sous-tâche qui correspond au corps de boucle. Dans ce cas, nous allons affecter l'indice de boucle à la sous-tâche correspondant au corps de boucle. Cet indice est mis à jour à chaque itération.

La *tâche conditionnelle* permet de représenter les structures conditionnelles d'un code (*if-then-else*, *switch-case*, *etc.*). Chaque branche de l'arbre de décision est représentée par des sous-tâches, par exemple *if-task* et *else-task*. Nous allons numéroter les sous-tâches en fonction de leur indice de branche dans l'arbre de décision.

La *tâche en point* n'est pas à proprement parler une tâche. C'est une tâche uniquement introduite dans le modèle afin de pouvoir effectuer des traitements de pilotage en un point précis du code. Elle ne peut pas contenir d'autres tâches et par conséquent elle n'est pas hiérarchique comme les autres tâches.

La représentation d'une simulation SPMD avec ce modèle revient donc à définir une *tâche hiérarchique* qui décrit l'ensemble du code de la simulation et qui contient des sous-tâches qui décrivent plus finement la structure du code. Cette tâche sera nommée *tâche hiérarchique principale*

(THP). C'est une tâche composée qui est *unique* pour une simulation donnée. Ainsi, par abus de langage, nous parlerons d'une *tâche hiérarchique principale* pour désigner la représentation complète de la simulation modélisée par cette tâche.

Afin d'illustrer nos propos, nous modélisons à l'aide d'un *MHT* une simulation SPMD que nous nommerons Π. La figure 2.3 montre la *THP* qui modélise la simulation Π. Cette simulation est un code composé de trois grandes parties : l'initialisation, la boucle de calcul et la terminaison. La boucle de calcul est elle-même subdivisée en trois sous-parties : un bloc conditionnel, une sous-boucle et un point d'interaction.

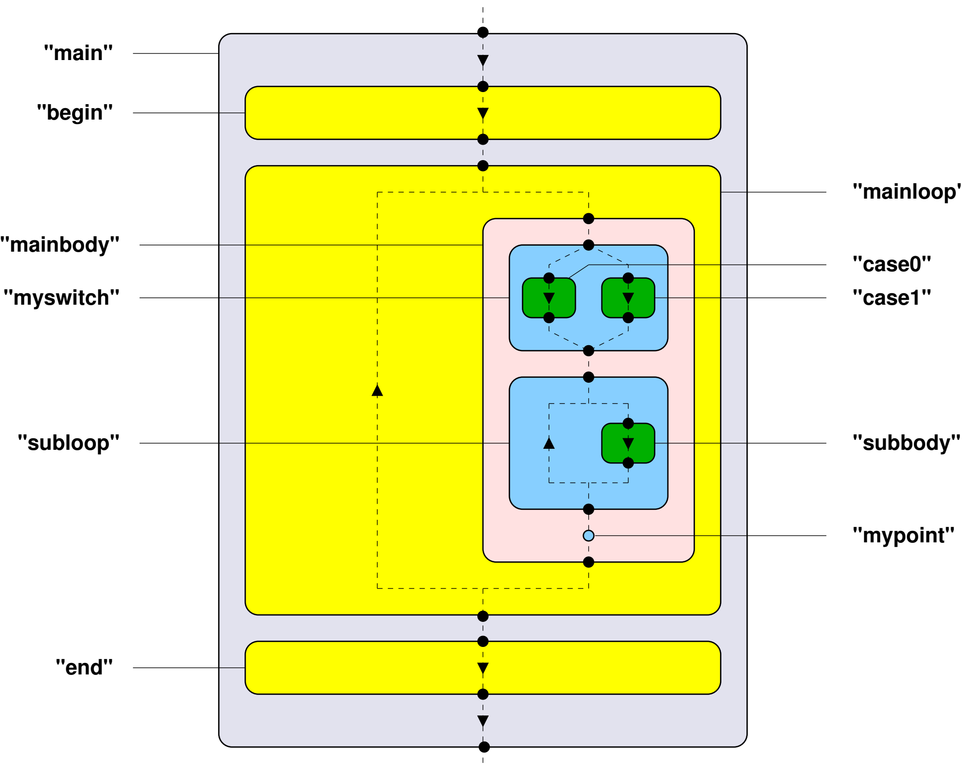


FIGURE 2.3 – Exemple d'une tâche hiérarchique principale représentant la simulation SPMD Π

La figure 2.4 représente l'arbre des tâches associé à la *THP* Π et met en évidence les relations de parenté entre les différentes tâches (tâches-mères/filles, tâches-sœurs). Cette représentation, contrairement à la précédente, ne donne pas d'indication sur le flot d'exécution, mais elle extrait la structure hiérarchique du code. Les indices au dessus de chaque nœud de l'arbre correspondent à la numérotation introduite dans la description des types de tâches. Les tâches *mainbody* et *subbody* ne sont présentes qu'une seule fois sur le schéma avec des indices i et j. En réalité, nous aurions dû les faire apparaître autant de fois qu'il y a d'itérations pour les boucles *mainloop* et *subloop*.

Nous avons présenté le modèle de représentation du flot d'exécution des simulations SPMD utilisé dans EPSN. Dans ce modèle, nous définissons une numérotation des tâches dans l'arborescence. Nous allons à présent voir comment nous utilisons ces indices de tâche afin de définir un système de date.

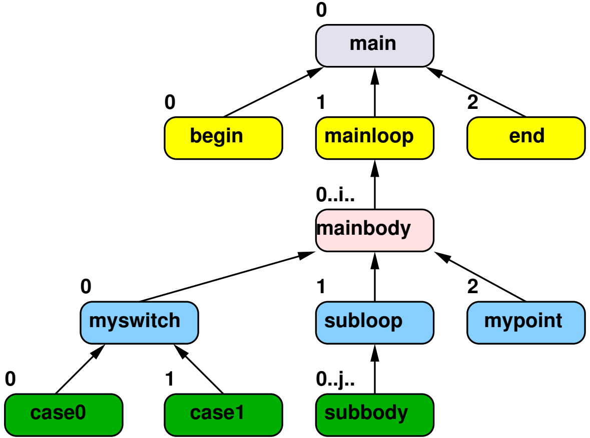


FIGURE 2.4 – Représentation en arbre de la *tâche hiérarchique* Π

2.2.1.2 Système de date

Afin de nous repérer de façon exacte dans le flot d'exécution, nous définissons un système de date. Les dates nous permettent de nous repérer avec précision dans le flot d'exécution, mais nous permettent également d'assurer la cohérence des traitements de pilotage, ce que nous verrons par la suite à la section 2.2.2.

Le système de date de EPSN s'appuie intégralement sur le *MHT*. Dans une *THP*, les tâches sont délimitées par deux points d'instrumentation, marquant le début et la fin de la tâche. Ainsi, nous distinguerons deux types de dates, les *dates de tâches* et les *dates de points*. Ces dates permettent donc de repérer, respectivement, des tâches et des points d'instrumentations dans le flot d'exécution.

Définition 1 (Date de tâche)
Une date de tâche d est un mot sur l'alphabet des entiers naturels $\mathbb{N}$, noté $d = d_0.d_1 \ldots d_{n-1}$ avec $\|d\| = n$ la longueur du mot.

Les dates de tâches sont construites de façon hiérarchique en définissant pour chaque niveau de la hiérarchie du THP un indice d_i dans la date. Ainsi, la date t de la tâche T est la concaténation de la date t_m de la tâche mère T_m de T et de l'indice k de la tâche T (i.e. $t = t_m.k$). L'indice k est le même indice que celui défini précédemment dans la description des tâches et dans l'arbre de tâches. La date d'une tâche T correspond donc à la concaténation des indices du chemin dans l'arbre des tâches de la racine au nœud de la tâche T.

Par la suite, l'ensemble des dates de tâches sera noté $\mathcal{D}_T$. Dans [40], l'auteur définit également une relation d'ordre strict et partiel [3] permettant de comparer les dates de tâches. Une date de tâche permet de repérer précisément une tâche. Mais ces dates ne sont pas suffisantes pour se repérer précisément dans le flot d'exécution. En effet, une date ne donne pas d'information sur la position dans cette tâche. Nous ne pouvons donc pas savoir par la simple connaissance d'une date de tâche si l'exécution se situe en début ou en fin de tâche et donc si elle est avant ou après une tâche fille. Par conséquent, pour nous repérer plus précisément dans le flot d'exécution, nous introduisons également des dates sur les points qui délimitent les tâches, en s'appuyant sur les dates de tâches que nous venons de définir.

3. Une relation d'ordre $<$ strict et partiel : soit deux dates $v = v_0.v_1 \ldots v_{n-1}$ et $w = w_0.w_1 \ldots w_{m-1}$ dans $\mathcal{D}_T$, on définit : $v < w$ si et seulement si $v_k < w_k$ où k est le plus petit indice tel que $v_k \neq w_k$

Définition 2 (Date de point)
Une date de point est une paire de la forme (t, p) *constituée d'une date de tâche* $t \in \mathcal{D}_T$ *et d'un entier* $p \in \{0, 1\}$ *marquant le début* 0 *ou la fin* 1 *de la tâche associée à* t*.*

On notera $\mathcal{D}_P = \mathcal{D}_T \times \{0, 1\}$ l'ensemble des dates de points. Nous utilisons les notations $b = 0$ et $e = 1$ pour délimiter un point de début (*begin*) ou un point de fin(*end*) (à ne pas confondre avec les tâches en point pour lesquelles les points de début et de fin de tâche sont confondus). Pour ces dates, nous avons une relation d'ordre strict et total [4]. Nous pouvons donc classer toutes les dates de points. Cela n'a de sens que pour des dates de points définies sur une même *THP*. Les dates de points étant associées à des points d'instrumentation, on peut donc à la simple connaissance des dates de points se repérer précisément dans le flot d'exécution. Mais nous pouvons également savoir si le passage sur un point d'instrumentation est antérieur à un autre dans le flot d'exécution.

Par la suite, lorsqu'on parlera d'une date, sans plus de précision, il sera sous-entendu qu'il s'agira d'une date de point.

2.2.1.3 Modèle de description des données

Outre la description d'une simulation en tâches, son pilotage passe aussi par l'interaction avec ses données. Nous devons donc avoir un moyen de spécifier les données utiles pour le pilotage. Ces données sont les données que l'on souhaite visualiser ou bien modifier lors du pilotage d'une simulation. Pour cela, nous allons en définir une modélisation simple.

Une donnée est représentée par un *support géométrique* (maillage non structuré, point, grille structurée, *etc.*) auquel sont associées des *variables*. À ces données multi-variées, nous allons également attribuer un *nom* ou identifiant, des *informations d'accessibilité* et une *révision*.

Les informations d'accessibilité sont définies pour toutes les tâches, c'est-à-dire que pour toutes les données, nous devons définir les droits d'accès au niveau de chaque tâche. Nous considérons différents contextes possibles permettant de préciser si la donnée est accessible en lecture (*readable*), modifiable (*writable*), ou bien protégée (*protected*) afin de ne pas être lue ou modifiée. En plus de ces droits d'accès, un contexte particulier permet de spécifier qu'une nouvelle *révision* de la donnée est générée (*modified*). Ces contextes d'accès sont définis de manière hiérarchique ; ainsi, si le contexte d'une tâche n'est pas spécifié, elle héritera du contexte de sa tâche mère.

Définition 3 (Révision de donnée)
La révision d'une donnée est la date de tâche à laquelle cette donnée a été produite.

La révision correspond donc toujours à la date d'une tâche ayant un contexte d'accès *modified* pour la donnée considérée. Lorsque la *révision* d'une donnée change, cela veut dire qu'il y a une nouvelle version de la donnée à visualiser. En cours d'exécution, différentes révisions sont produites et nous distinguerons la révision la plus récente d'une donnée comme étant sa *révision courante*. Une donnée peut donc avoir plusieurs révisions, toutes les révisions d'une donnée définissent un ensemble de révisions. Toutes les révisions de cet ensemble ne sont pas forcément utiles à visualiser. Nous allons donc définir un masque de date permettant de restreindre un ensemble de dates ou de révisions. Cela nous permet donc de définir des ensembles de révisions « *utiles* ».

4. Relation d'ordre strict et total : pour tout couple de date v et w de $\mathcal{D}_P$ on peut définir $v < w$, $v = w$ ou $v > w$.

Définition 4 (Masque de date)
Un masque de date est un mot de l'alphabet $\mathbb{N} \cup \widetilde{\mathbb{N}}$, *avec* $\widetilde{\mathbb{N}} = \{\widetilde{k} \mid k \in \mathbb{N}\}$.

Définition 5 (Ensemble de dates restreint à un masque)
Soit $m = m_0.m_1 \dots m_{n-1}$ *un masque de date, on définit l'ensemble des dates restreintes au masque* m, *noté* $\mathcal{D}_T(m)$, *comme étant toutes les dates appartenant à*
$\gamma(m) = \gamma(m_1).\gamma(m_2).\cdots.\gamma(m_n)$ *tel que*

$$\forall i \in [1, n],\ \gamma(m_i) = \begin{cases} \{k\} & si\ m_i = k \\ \{kp\ avec\ p \in \mathbb{N}\} & si\ m_i = \widetilde{k}. \end{cases}$$

Par exemple le masque $0.3.\widetilde{2}.4$ permet de définir l'ensemble des dates de tâche $\mathcal{D}_T(0.3.\widetilde{2}.4) = \{0.3.0.4, 0.3.2.4, \cdots, 0.3.2n.4\}$. Nous pouvons grâce à cela restreindre l'ensemble des révisions d'une donnée à l'ensemble des révisions générées une itération sur deux.

Ce masque nous permet donc de définir l'ensemble des révisions d'une donnée. Par exemple, si nous considérons une donnée D qui est mise à jour dans la tâche *myswitch* du *THP* Π présenté précédemment, on peut construire le masque $m_D = 0.1.\widetilde{1}.0$. Ainsi l'ensemble $\mathcal{D}_T(m_D) = \{0.1.1.0, 0.1.2.0, \cdots, 0.1.n.0\}$ définit l'ensemble des révisions possibles de la donnée D, que l'on notera $\mathcal{R}_D(\Pi)$. Dans ce cas, une nouvelle révision est générée à chaque itération de la boucle principale qui a pour date $0.1.i.0$. Dans le cas où plusieurs tâches avec le contexte d'accès *modified* existent pour une même donnée, nous définissons un masque de date pour chacune de ces tâches et l'ensemble des révisions de la donnée sera la concaténation des ensembles associés à ces masques. Cette notion de masque de date a été introduite dans le modèle au cours des travaux de cette thèse afin de définir plus finement la cohérence des traitements des requêtes de données. Mais elle servira essentiellement dans le cas de simulations distribuées.

2.2.1.4 Exemple

Afin de mieux illustrer le modèle de description dans son intégralité, nous considérerons une simulation SPMD et nous allons présenter sa modélisation. Prenons par exemple une simulation parallèle de diffusion de la chaleur que nous nommerons Γ.

Cette simulation résout l'équation de la chaleur en deux dimensions par la méthode des différences finies en espace et en temps en utilisant un schéma d'Euler explicite. La simulation calcule en boucle la température au temps $t+1$ en chaque point d'une grille 2D à partir des valeurs de la température à l'itération précédente (au temps t) $u_{t+1} = f(u_t)$. L'itération précédente est stockée dans une autre grille. À chaque itération, lorsque les nouvelles valeurs de la température sont calculées, les deux grilles sont inversées $u_t = u_{t+1}$. La nouvelle grille devient l'ancienne et l'ancienne est écrasée par les nouvelles valeurs calculées. Le découpage en tâches est donc le suivant : une tâche d'initialisation de la grille (*init*), puis une boucle de calcul (*timeloop*) qui contient deux tâches : une tâche de mise à jour de la température (*diffusion*) et une tâche d'échange des grilles (*swap*), les 2 encapsulées dans la tâche du corps de boucle (*timebody*).

Dans cette simulation, la donnée intéressante pour le pilotage est la grille de température. Il s'agit d'une grille structurée 2D à laquelle est associée à chaque point une variable contenant la *température* (un tableau de réels double précision). Si nous utilisons la grille contenant l'ancien pas de temps, elle est lisible quasiment tout le temps. La variable de température est modifiée sur cette grille lors de la tâche *init* et périodiquement dans la tâche *swap*. De base, nous avons aucune tâche durant laquelle nous pouvons écrire les données. Par conséquent, pour se laisser la possibilité de modifier les données de la simulation, nous rajoutons une tâche en point pour laquelle nous autorisons l'écriture de la donnée (contexte *writable*).

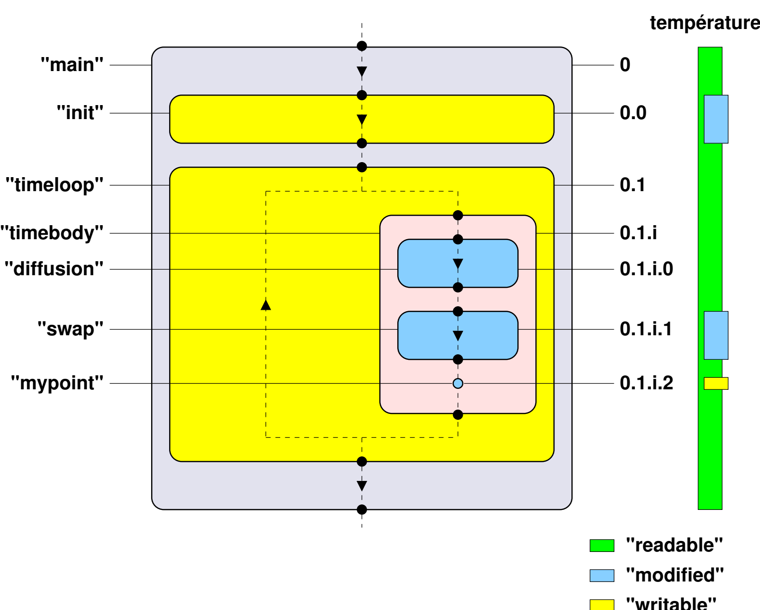


FIGURE 2.5 – Modélisation de la simulation de diffusion de la chaleur Γ

Nous pouvons donc définir l'ensemble des révisions de la *température* à partir des dates des tâches *init* et *swap* : $\mathcal{R}_{temperature}(\Gamma) = \mathcal{D}_T(0.0) \cup \mathcal{D}_T(0.1.\tilde{1}.1) = \{0.0, 0.1.1.1, 0.1.2.1, \cdots, 0.1.n.1\}$. Dans cette simulation il n'y a pas de contrainte spéciale sur les données ; par conséquent l'ensemble des révisions $\mathcal{R}_{temperature}(\Gamma)$ définit les révisions que l'on peut récupérer dans la simulation pour les visualiser. Dans une simulation comme celle-ci les masques des dates servent donc juste à définir l'ensemble de toutes les révisions d'une donnée et non pas à restreindre l'ensemble des révisions à un sous-ensemble.

2.2.2 Modèle de pilotage

Les interactions avec les simulations sont décrites dans la seconde partie du modèle abstrait, ce que nous avons appelé modèle de pilotage dans l'introduction de ce chapitre (voir fig. 2.1). Ce modèle de pilotage est composé de plusieurs sous-parties. Tout d'abord nous décrirons les interactions possibles avec la simulation sous la forme de différents types de requêtes. Puis nous aurons les algorithmes de coordination permettant d'assurer la cohérence du traitement de ces requêtes. Dans la suite de cette section, nous présentons ces deux aspects du modèle de pilotage.

2.2.2.1 Pilotage par les requêtes

La plateforme EPSN est basée sur un concept de pilotage par les requêtes : ce sont des requêtes venant d'un client de pilotage qui déclenchent un traitement au niveau de la simulation. Tant qu'aucune requête n'est reçue, la simulation suit son cours normalement sans être perturbée par le pilotage. Cette approche s'oppose à d'autres méthodes de pilotage comme le pilotage par flux d'évènements. Dans ce dernier cas, la simulation émet systématiquement des évènements de pilotage, par exemple lorsque de nouvelles données à visualiser ont été générées. D'une manière générale, la difficulté lors du traitement d'une requête réside dans le fait d'assurer la cohérence du résultat car la requête est traitée de façon asynchrone par tous les processus de la simulation. Il faut donc déterminer une date de traitement afin d'assurer une prise en compte

au même « *moment* » par tous les processus. Cela peut être fait de différentes manières, soit en synchronisant fortement tous les processus d'une simulation avant le traitement, soit en utilisant des techniques de planification afin de prévoir à l'avance la date de traitement. Dans EPSN, nous avons opté pour la planification des traitements car cette approche est moins intrusive ; de plus, tant qu'aucune requête n'est reçue , nous n'aurons aucune planification à effectuer.

Dans EPSN, nous avons différents types de requêtes. Tout d'abord nous avons les requêtes de contrôle (*pause*, *step*, *play*) permettant de contrôler le flot d'exécution de la simulation, le mettre en pause, le faire avancer de point d'instrumentation en point d'instrumentation, ou le faire continuer s'il avait été arrêté. Puis nous avons les requêtes de données (*get*, *put*) qui permettent de récupérer les données depuis un client ou de les modifier dans la simulation.

Cohérence La *cohérence* du résultat d'une requête dépend donc de la date du traitement, mais d'autres critères rentrent également en compte. Nous appellerons ces critères : *conditions locales*. Ce sont des conditions devant être vérifiées localement sur chaque processus de la simulation. Cette vérification locale est faite indépendamment sur tous les processus. Les *conditions locales* dépendent du type de la requête à traiter, de contrôle ou de données. Nous allons préciser les *conditions locales* pour ces deux types de requêtes.

1. Pour les requêtes de contrôle, la *condition locale* consiste uniquement à vérifier que la date de traitement planifiée est atteinte.
2. Pour les requêtes de données, il faut, comme pour les requêtes de contrôle, vérifier que la date de traitement est atteinte. Mais il faut aussi vérifier que la révision de la donnée est correcte, c'est-à-dire qu'elle est la même sur tous les processus et qu'elle est bien en accord avec le masque de date associé à la requête (s'il existe). Il faut également vérifier que les droits d'accès sont en adéquation avec la requête (lecture/écriture). Les droits d'accès sont ceux que nous avons définis précédemment (voir section 2.2.1.3).

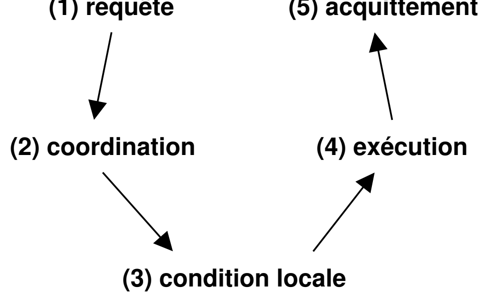


Figure 2.6 – Cycle de vie d'une requête

Le cycle de vie d'une requête est illustré par la figure 2.6. Suite à la réception d'une requête au niveau de la plateforme de pilotage côté simulation (étape 1) débute une phase de coordination (étape 2) qui est basée sur un algorithme de planification. Le but est de déterminer une *date planifiée* t_p pour exécuter de manière cohérente la requête sur tous les processus de la simulation. Une fois cette date déterminée, il reste à évaluer la *condition locale* (étape 3). Cela consiste au niveau de chaque processus à vérifier que la date t_p est atteinte et que les différents accès sont en accord avec la requête. Une fois ces conditions vérifiées, les processus peuvent exécuter la requête (étape 4). Les processus exécutent la requête indépendamment les uns des autres. Cette exécution doit être réalisée tant que la *condition locale* reste vraie. Si cette condition n'est plus

vérifiée au cours de l'exécution, le flot d'exécution de la simulation doit être momentanément suspendu sur le point d'instrumentation où la condition n'est plus vérifiée. Une fois l'exécution de la requête terminée, l'exécution de la simulation peut reprendre son cours. Ces précautions sont prises afin de ne pas remettre en cause la cohérence de la requête. Cela pourrait arriver par exemple dans le cas où l'exécution arriverait au niveau d'un point d'instrumentation marquant le début d'une tâche où la donnée que l'on souhaite récupérer serait modifiée alors que la requête demandant la donnée n'a pas fini d'être traitée. La fin de la requête est notifiée par une phase d'acquittement (étape 5) entre la simulation et le client.

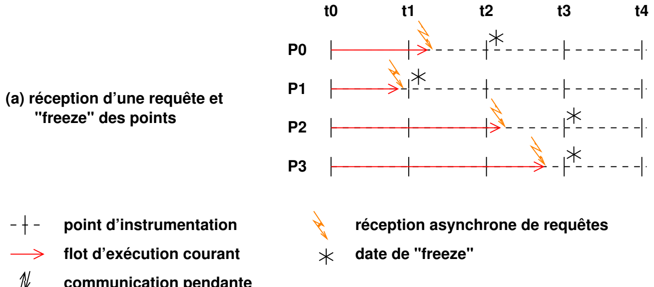
Pour permettre le suivi au cours du temps de l'évolution d'une donnée, nous avons introduit une requête un peu spéciale : la requête de données *permanente*. Contrairement à une requête de données simple, qui ne récupère la donnée qu'une seule fois, les requêtes de données permanentes récupèrent la donnée en permanence c'est-à-dire dès qu'une nouvelle révision est produite. Ces requêtes sont paramétrables et nous pouvons choisir la périodicité des révisions que nous souhaitons récupérer. Ces requêtes s'exécutent donc plusieurs fois mais elles ne nécessitent qu'une seule phase de coordination. Pour être *cohérentes*, ces requêtes se comportent comme une requête de données simple à la première exécution. Mais par la suite, une fois la planification effectuée il suffit de vérifier que la nouvelle révision de la donnée a bien été augmentée d'un nombre d'itération correspondant à la périodicité spécifiée dans la requête et qu'elle est en adéquation avec le masque quand celui-ci est défini.

La principale difficulté dans ce processus de traitement des requêtes est l'algorithme de coordination. Il doit définir une date qui permet à tous les processus d'exécuter la requête de façon cohérente en temps. Nous allons maintenant détailler cet algorithme.

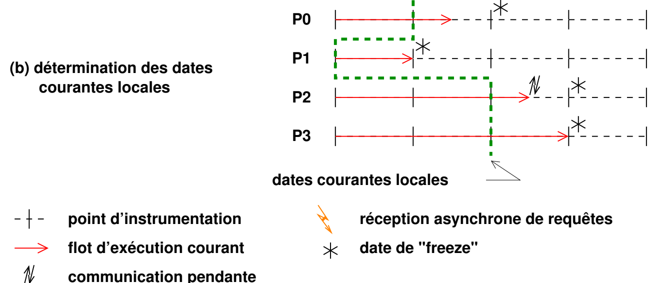
2.2.2.2 Algorithme de planification dans le cas SPMD

L'algorithme de planification SPMD débute au moment de la réception de la requête sur les processus. Il se découpe en quatre étapes principales :

Étape 1 Tous les processus « *gèlent* » tous leurs points d'instrumentation : cela signifie que lors de leurs exécutions, ils ne dépasseront pas le prochain point d'instrumentation rencontré. Nous appellerons cette étape, l'étape de *freeze*.

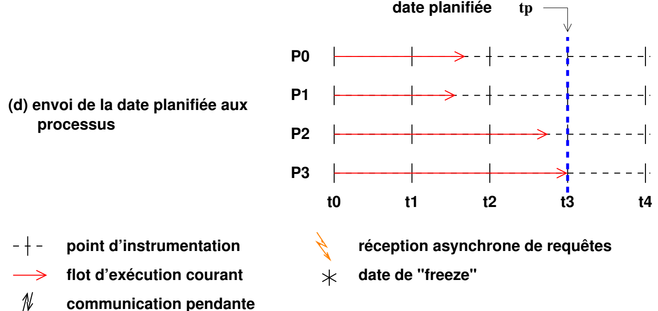


Étape 2 Tous les processus déterminent leur *date courante*. Cette date correspond à la date locale du processus et n'est pas nécessairement la même pour tous les processus qui avancent de manière plus ou moins synchronisée. Par exemple, pour le processus $P0$, la date courante est t_1 alors que pour le processus $P1$ c'est t_0. Ces dates sont envoyées à un *coordinateur*.



Étape 3 Une fois que le *coordinateur* a obtenu les *dates courantes* de tous les processus, il va déterminer la *date planifiée* t_p. Cette date est la date immédiatement supérieure à la date courante maximale (t_{max}). Dans notre cas cette date est t_3 car $t_{max} = t_2$.

Étape 4 La *date planifiée* est transmise par le coordinateur à tous les processus. Les processus relâchent l'ordre de *freeze* à la réception de cette date.



Durant la première étape de cet algorithme, nous pouvons temporairement créer un inter-blocage entre plusieurs processus. Par exemple, sur le schéma de l'étape 2, $P2$ à une communication pendante vers $P1$ qui est bloqué sur un point d'instrumentation. Toutefois cet état n'est que temporaire car à l'étape 4, tous les points gelés sont relâchés. Par ailleurs, si la *date planifiée* est déterminée avant que les processus atteignent un point d'instrumentation gelé, la simulation n'est pas stoppée et dans ce cas le calcul de la date est totalement recouvert.

À la fin de cet algorithme, tous les processus connaissent la *date planifiée* et pourront donc effectuer leurs traitements après vérification de la *condition locale*.

2.2.2.3 Exemple

Nous reprenons l'exemple de la simulation de la diffusion de la chaleur Γ afin d'illustrer le modèle de pilotage. Nous pouvons par exemple durant l'exécution de la simulation sur six processeurs, mettre en pause l'exécution de cette dernière ou encore récupérer les données concernant la température afin de les visualiser.

Si l'exécution courante se trouve au début de la 1338$^{\text{ième}}$ itération et que nous voulons la mettre en pause, une requête de contrôle est envoyée à la simulation et l'algorithme de coordination est exécuté. Les points d'instrumentation sont donc gelés, ce qui ne signifie pas que la simulation ne continue pas son exécution. Dans cette simulation, les processus ne peuvent pas être totalement désynchronisés car il y a des phases de communication entre les processus voisins dans la tâche *diffusion* afin d'échanger les valeurs aux frontières de la grille. Ainsi, les dates de point des processus se situent entre 0.1.1336.0.b et 0.1.1337.0.b, car on ne peut pas finir la tâche *diffusion* si les autres processus ne l'ont pas atteint. Si nous considérons que le processus le plus avancé est juste au début de la tâche de mise à jour de la température (*diffusion*), soit à la date 0.1.1337.0.b, la *date planifiée* sera donc la date immédiatement supérieure rencontrée dans le

flot d'exécution, soit $0.1.1337.0.e$ (la fin de la tâche *diffusion*). Dès que les processus atteindront cette date, ils mettront leur exécution en suspens, c'est-à-dire sur le point d'instrumentation marquant la fin de la tâche *diffusion*.

Si maintenant, dans la même configuration, au lieu de demander une mise en pause de l'exécution, nous demandons à récupérer la donnée *température*, la même phase de planification aura lieu. Mais au lieu de suspendre leur exécution, une fois la date atteinte, les processus vont vérifier les droits d'accès de la donnée. Dans notre cas, la donnée est immédiatement *readable*, par conséquent l'envoi de la donnée va débuter, en concurrence de l'exécution de la simulation et indépendamment sur chaque processus. Si un processus atteint le début de la tâche *swap* avant d'avoir fini son envoi de données, son exécution sera mise en suspens jusqu'à la fin de l'envoi de la *température*. En effet, dans la tâche *swap* la donnée est marquée *modified* et par conséquent elle n'est plus accessible. Si les *conditions locales* ne sont pas vérifiées immédiatement, le traitement de la requête reste en attente jusqu'à ce qu'elles le deviennent. Le caractère SPMD de la simulation fait que si elles sont vérifiées à une certaine date pour un processus, elles le seront à la même date pour tous les autres processus. Si ces conditions n'étaient jamais vérifiées, la simulation finirait son exécution sans que la requête puisse être traitée, d'où l'importance d'une bonne modélisation.

Dans le cas d'une requête permanente avec une fréquence de 1, la requête sera ré-exécutée après la production de chaque nouvelle révision, soit juste après la tâche *swap*, à la date $0.1.1337.1.e$ où la donnée redevient *readable*. Elle sera de plus récupérée à toutes les itérations suivantes, donc à toutes les dates de point correspondant au masque $0.1.\widetilde{1}.1.e$ et plus grandes que la date courante.

2.2.3 Limitations du modèle pour les simulations distribuées

Ce modèle a été conçu pour décrire et piloter des simulations SPMD ; il est donc prévu pour des simulations qui ont un flot d'exécution unique pour tous les processus de la simulation. Dans les simulations distribuées, tous les processus ne suivent pas forcément le même programme, et ce que ce soit pour des simulations Client/Serveur ou des simulations M-SPMD. Ces simulations sont typiquement des couplages de plusieurs codes. Par conséquent, il n'est pas facile de les représenter avec une seule *THP* tel que nous l'avons défini précédemment. Ces codes contiennent une boucle en temps explicite ou implicite (voir section 1.1.2). Il est possible de décrire les codes de simulation contenant une boucle en temps explicite à l'aide d'une *THP*, comme les codes clients des simulations Client/Serveur ou les différents codes SPMD d'une simulation M-SPMD. Mais cela ne suffit pas à représenter la simulation distribuée dans sa globalité ; en effet, dans le cas des simulations Client/Serveur, les appels de méthode à distance ne sont pas prises en compte par le modèle, et dans le cas des simulations M-SPMD c'est l'unicité de la simulation qui n'est pas représentée. Cela est dû au fait qu'on ne peut pas décrire des tâches différentes et exécutées de façon simultanée dans une même simulation couplée. La modélisation d'une simulation par une *tâche hiérarchique principale* n'est donc plus adaptée au cas des simulations distribuées que nous visons. Il faudra donc un autre moyen pour décrire ces simulations, ce qui va également impliquer des modifications dans le système de dates. C'est donc tout le MHT qui devra être revu.

Cependant, les limitations ne sont pas simplement au niveau du *MHT*. En effet, les données que l'on souhaite visualiser dans une simulation distribuée sont bien plus complexes que dans des simulations parallèles. Dans le cas des simulations distribuées, chaque code possède ses propres données alors que les données utiles à la visualisation sont typiquement une composition de plusieurs données locales aux codes. Il faut donc un moyen de décrire ces données de haut niveau

constituées de plusieurs variables issues de données, et de codes différents ; nous avons décrit une telle situation dans le cas de la simulation LibMultiScale ou du couplage DLPoly/Siesta (voir section 1.1.2).

Pour finir, comme le modèle de description n'est pas suffisant pour représenter des simulations distribuées, le modèle de pilotage ne sera pas non plus suffisant. En effet, comme les descriptions du flot d'exécution et des données doivent être modifiées, la définition de la cohérence devra être modifiée en conséquence. Nous devrons donc revoir les algorithmes de planification qui permettent d'assurer la cohérence.

Nous allons donc devoir introduire un nouveau modèle abstrait pour décrire ces simulations distribuées.

2.3 Modèle abstrait pour les simulations distribuées

Comme nous l'avons vu précédemment, le modèle abstrait de EPSN n'est pas suffisant pour représenter des simulations distribuées. Nous allons donc enrichir le modèle abstrait de EPSN. Nous allons nommer notre nouveau modèle, *Modèle Hiérarchique en Tâches distribué (MHTd)*, contrairement au *MHT* décrit dans la section 2.2.1.1 que nous nommerons dorénavant *Modèle Hiérarchique en Tâches parallèle (MHTp)* afin de les différencier. Il est à noter que le *MHTd* est une extension du *MHTp* et on peut donc tout à fait modéliser des simulations parallèles avec le *MHTd*. Dans la suite de cette section, nous allons décrire les modifications apportées aux différentes parties du modèle abstrait.

2.3.1 Modèle de description de simulations distribuées

Tout comme dans le cas des simulations parallèles, le modèle de description se divise en trois parties : (1) la modélisation du flot d'exécution, (2) la définition d'un système de dates et (3) la modélisation des données.

2.3.1.1 Modèle hiérarchique en tâches distribué (MHTd)

Comme on l'a montré dans la section 2.2.3, ce modèle est limité par construction aux simulations SPMD. Par conséquent, nous allons devoir l'adapter afin de prendre en compte des simulations plus complexes à savoir les codes M-SPMD ou les codes Client/Serveur.

Si l'on prend l'exemple d'une simulation Client/Serveur, on a un code client contenant une boucle en temps principale, et des codes serveurs contenant les services distants utilisés par le client. Si nous voulons représenter une telle simulation avec un *MHTp*, nous pouvons représenter le client mais nous ne pouvons pas représenter les serveurs, car ce modèle ne permet pas de représenter les appels à distance ainsi que les fonctions ou services fournis par les serveurs.

Les simulations M-SPMD et Client/Serveur (voir section 1.1.2) sont composées de plusieurs codes. Mais elles ont en commun le fait d'avoir une « *boucle en temps* » qu'elle soit implicite ou explicite. Nous allons donc modéliser cette boucle en temps avec une tâche hiérarchique. Pour représenter le reste des codes, nous allons utiliser d'autres tâches hiérarchiques que l'on va devoir définir. L'assemblage de codes composant les simulations couplées va donc être modélisé par un assemblage de tâches hiérarchiques, globales et locales.

Définition 6 (Tâche hiérarchique globale)
Une tâche hiérarchique globale *(THG) est une tâche hiérarchique qui explicite le flot d'exécution global d'une simulation distribuée, c'est-à-dire le flot d'exécution de haut niveau commun aux différents codes.*

Une *THG* est donc la *tâche hiérarchique principale* que décrit une simulation distribuée. Par définition, c'est donc la tâche hiérarchique qui va représenter la boucle principale d'une simulation, que cette boucle soit explicite ou pas.

Définition 7 (Tâche hiérarchique locale)
Une tâche hiérarchique locale *(THL) est une tâche hiérarchique décrivant les parties spécifiques à chaque code d'une simulation distribuée.*

On définit une *forêt de tâches hiérarchiques locales* (FTHL) comme étant un ensemble de *THL* associées à un même code. Une FTHL permet par exemple de décrire les différents services fournis par un code serveur, chaque service étant représenté par une THL.

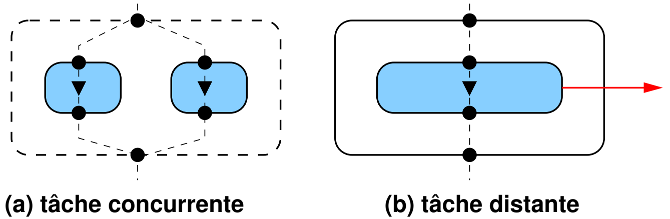


FIGURE 2.7 – Les différents types de tâches utilisées dans le MHT distribué

Les quatre tâches de base du *MHT parallèle* ne sont pas suffisantes pour décrire toute la complexité d'une simulation distribuée. Il nous faut donc introduire de nouveaux types de tâches afin de prendre en compte les spécificités des simulations distribuées. Ces nouvelles tâches sont aux nombres de deux, les *tâches concurrentes* (Fig. 2.7.a) et les *tâches distantes* (Fig. 2.7.b).

- La *tâche concurrente* est une tâche englobant plusieurs tâches qui s'exécutent de façon simultanées. Ainsi nous pouvons décrire plusieurs tâches dont l'exécution est faite de manière concurrente, dans des codes différents, en les englobant dans une *tâche concurrente*. Il est a noter que même si la représentation de ces tâches est similaire à celle des tâches conditionnelles, elles n'ont bien sûr rien à voir.
- La *tâche distante* est une tâche permettant de représenter un appel de méthode à distance. Ainsi la description détaillée de la tâche distante, la *THL*, se trouve dans la *FTHL* propre au code sur lequel l'appel est effectué. Cette tâche permet donc de relier une *THG* à une *THL*.

Après avoir défini les nouveaux types de tâches utilisées pour le *MHTd*, nous pouvons décrire toute la complexité d'une simulation distribuée. Les figures 2.8 et 2.11 illustrent des exemples de simulation dans lesquelles nous avons des tâches qui s'exécutent à distance, mais également de façon simultanées. Nous avons donc tout ce qu'il nous faut pour décrire des simulations Client/Serveur et M-SPMD.

Simulation Client/Serveur – Pour une simulation Client/Serveur, comme la simulation Δ (Fig. 2.8), la similitude entre le modèle de programmation et le modèle de représentation dans le *MHTd* est assez intuitif. Le client Δ va effectuer des appels de méthode sur différents serveurs Δ_0 et Δ_1, ce qui se décrit de façon naturelle par des *tâches distantes*. Ainsi le code client, qui contient la boucle en temps principale, est décrit par une *THG*. Les appels à distance sur les méthodes du serveur sont représentés par des *tâches distantes* pointant vers des *THL* décrites dans les *FTHL* modélisant les codes serveurs. Dans ce cas, les *FTHL* remplacent la *THG* pour décrire les codes serveurs, car plusieurs THL sont souvent nécessaires pour réaliser la description d'un serveur.

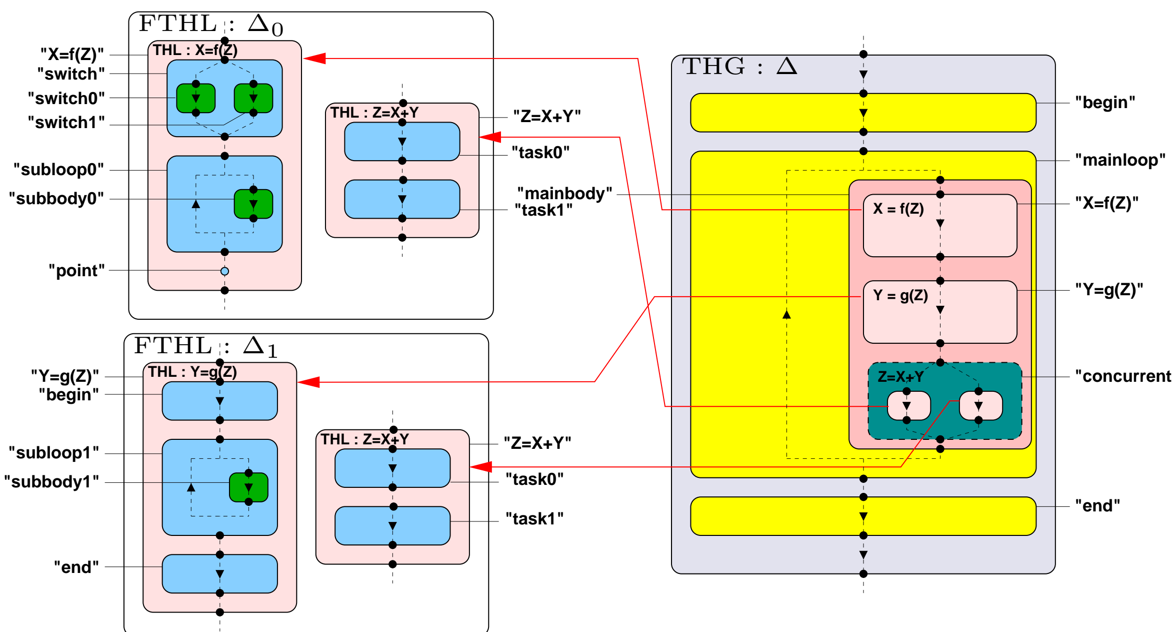


FIGURE 2.8 – Exemple de modélisation d'une simulation Client/Serveur Δ. Le client à droite fait des appels de méthodes sur les serveurs Δ_0 et Δ_1 à gauche

Afin de mieux comprendre cette modélisation, nous décrivons la représentation de la simulation Δ illustrée en figure 2.8. Cette simulation nous servira d'exemple tout au long de cette section. Cet exemple permet d'avoir un aperçu des deux nouveaux types de tâches, ainsi que de la division de la description dans une *THG* et des *THL*. L'exemple Δ est donc constitué d'une *THG*, composée de deux tâches principales *begin* et *end* et d'une boucle principale *mainloop* représentant la boucle en temps explicite de la simulation. Cette boucle est constituée d'un corps de boucle *mainbody*. Cette description correspond à ce que l'on peut trouver dans la description d'un code SPMD classique. Comme nous allons le voir, c'est la composition de la tâche *mainbody* qui est propre à un code distribué. Ce corps de boucle contient trois sous-tâches, de types *distantes* et *concurrentes*. Les deux premières sous-tâches sont des tâches distantes qui pointent respectivement vers les *THL* *X=f(Z)* et *Y=g(Z)* dans les *FTHL* Δ_0 et Δ_1. La troisième tâche est une tâche concurrente indiquant que les deux tâches distantes *Z=X+Y* des *FTHL* Δ_0 et Δ_1 s'exécutent simultanément.

La figure 2.9 reprend également l'exemple de la simulation Δ et donne la représentation en arbre de tâches de la *THG* et des *THL*. Ainsi on peut voir que les *tâches concurrentes* ne sont pas, à proprement parler, des tâches mais uniquement des contextes spécifiant la simultanéité de l'exécution des tâches contenues. De plus, les tâches distantes sont toujours des feuilles de l'arbre qui sont reliées à la racine d'un arbre distant représentant une *THL*.

On définit l'*arbre complet* représentant la simulation comme étant la fusion en un arbre unique de la *THG* et des *THLs* de la simulation.

Simulation M-SPMD – Pour le cas M-SPMD, la juxtaposition des modèles de programmation et de représentation n'est pas aussi évidente. Ceci est dû au fait que nous avons autant de boucles en temps que de codes dans la simulation. Par conséquent, on devrait avoir autant de *THG* que de codes, ce qui est en contradiction avec l'idée que la simulation distribuée a une évolution en temps unique, même si elle est composée de plusieurs codes (voir figure 2.10). Pour y remédier, nous allons représenter les simulations M-SPMD comme des codes Client/Serveur

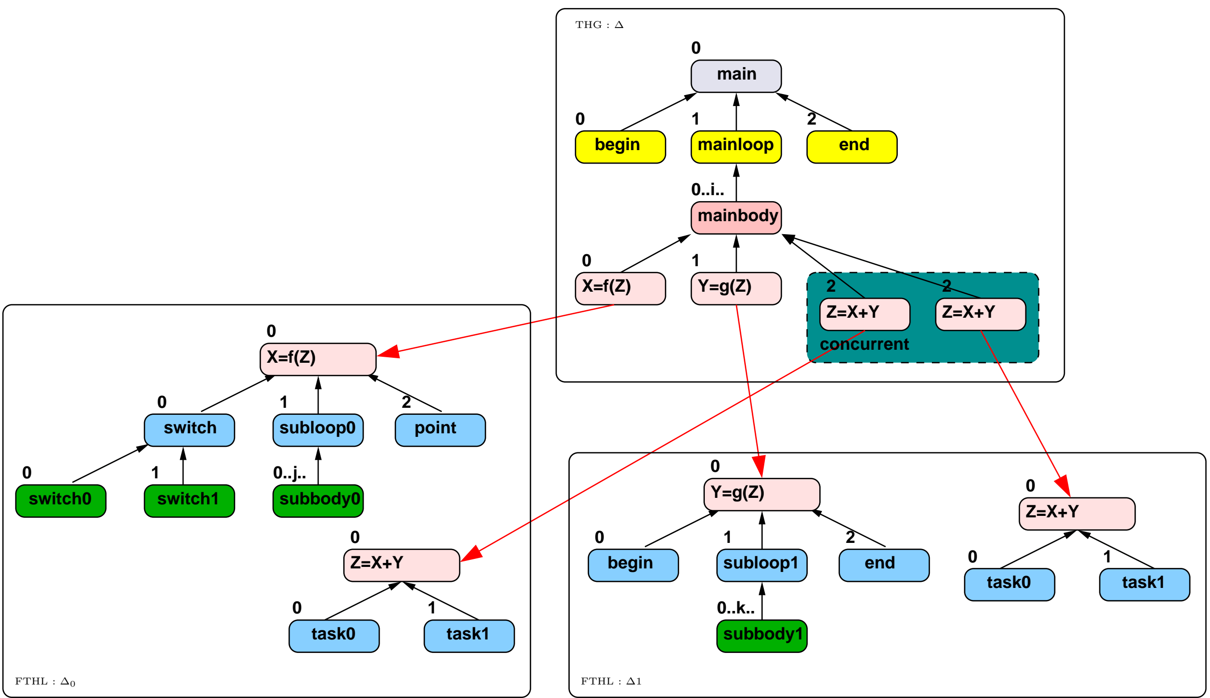


Figure 2.9 – Représentation en arbre de tâches de la simulation Δ

en explicitant la boucle en temps dans un *THG*. Dans ce *THG*, nous aurons des *tâches distantes* englobées dans des *tâches concurrentes* pour représenter les tâches réelles des codes. La *THG* représentera donc les parties similaires dans les différents codes, alors que les *THL* modéliseront les « *détails* » qui diffèrent entre les codes. Les *THL* ne sont donc pas ici des tâches indépendantes dans des serveurs, mais ce sont les sous-tâches qui étaient dans les *THP* représentant les codes SPMD. Cette représentation nous permet donc d'avoir un modèle de description du flot d'exécution unifié, que ce soit pour des simulations Client/Serveur ou M-SPMD, et ce qui sera utile pour coordonner des opérations cohérentes en temps inter-codes.

Nous allons également prendre un exemple de simulation M-SPMD afin de mieux comprendre ce qu'est une *THG* dans ce cas précis. Pour cela, nous allons prendre l'exemple illustré par les figures 2.10 et 2.11. Dans cet exemple, les *FTHL* sont constituées de tâches hiérarchiques venant des *THP* décrivant les différents codes. La *THG* contient des *tâches concurrentes* à l'initialisation et à la finalisation. Ces *tâches concurrentes* sont composées de *tâches distantes* pointant vers les tâches réelles dans les *FTHL*. De plus, la *THG* contient une représentation de la boucle principale en temps. Cette représentation est également basée sur des *tâches concurrentes* et des *tâches distantes*. Dans la représentation de la figure 2.10, on a les *THP* des deux codes qui ont été couplés. Ainsi, on peut noter que si nous remplaçons les tâches concurrentes par les *THL* respectives de l'un ou l'autre des codes (*FTHL : Code A* ou *FTHL : Code B*) dans la modélisation de la figure 2.11, on retrouve la description des codes SPMD.

Ainsi, nous avons introduit un moyen de modéliser à la fois des codes Client/Serveur et et des codes M-SPMD, et ce grâce à une modélisation générique et unifiée, en enrichissant le modèle hiérarchique en tâches à l'aide de nouveaux types de tâches. Les *tâches distantes* et *concurrentes* permettent donc de définir des *tâches hiérarchiques locales* et *globales* représentant des simulations distribuées de façon générique. L'unification des modèles nous permet par la suite de définir un système de dates unique et d'assurer la cohérence en temps grâce à des

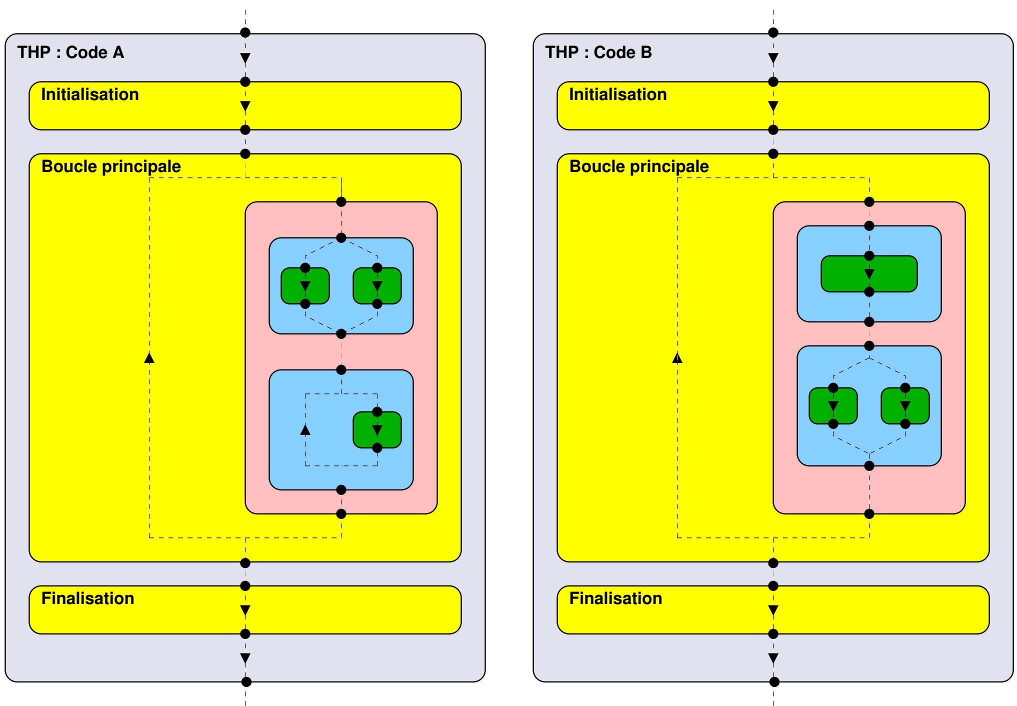


FIGURE 2.10 – Exemple de modélisation d'une simulation M-SPMD selon le *MHTp*. Chaque code SPMD peut être modélisé par une *THP*, mais ce modèle ne rend pas compte du couplage de ces codes

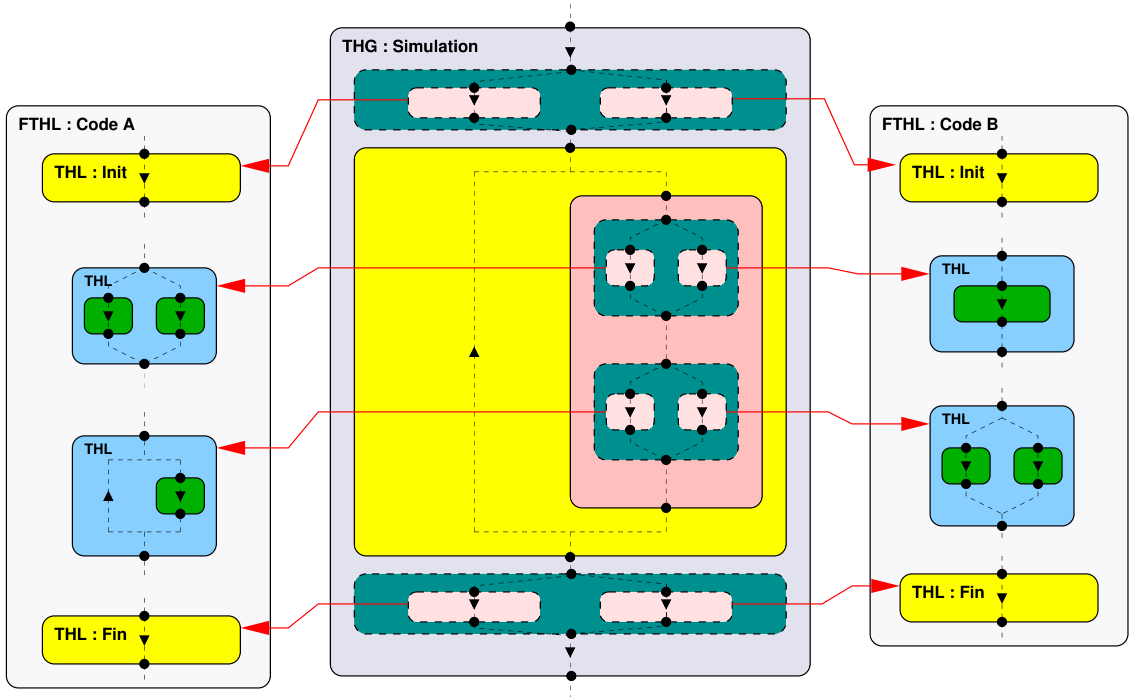


FIGURE 2.11 – Exemple de modélisation d'une simulation M-SPMD selon le *MHTd*

algorithmes communs pour les différents types de simulations.

2.3.1.2 Système de dates

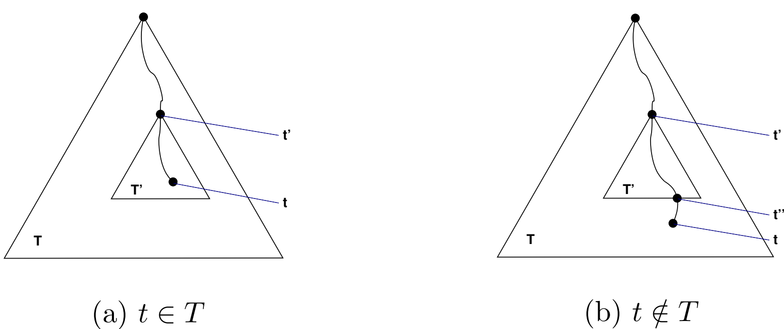
Comme nous l'avons vu dans le cas de simulations parallèles, il est possible de se repérer dans le flot d'exécution grâce au système de dates introduit dans la section 2.2.1.2. Il nous faut à présent enrichir ce système de dates afin de prendre en compte les nouveaux types de tâches.

En effet, le système de dates a été construit à partir de la représentation en tâches hiérarchiques introduit dans le *MHTp* ; il ne prend donc pas en compte les *tâches concurrentes* et *distantes*. Pour construire un système de dates dans le *MHTd*, il nous faut donc définir quelle est la date pour ces deux types de tâches.

Les *tâches concurrentes* sont des tâches qui s'exécutent simultanément, il nous faut donc un moyen d'exprimer cette simultanéité. Le moyen le plus simple et le plus intuitif est de donner la même date à toutes les tâches filles d'une tâche concurrente. Ainsi si nous reprenons la définition d'une date de tâche, nous allons noter la tâche T_m et ses T_{fi} tâches filles, avec i le numéro dans la séquence des tâches filles. Les tâches T_m et T_{fi} ont pour date de tâche respective t_m et t_{fi}. Si T_m était une *tâche composée*, on aurait donc $t_{fi} = t_m.i$. Pour une *tâche concurrente*, cela ne permet pas de représenter la simultanéité d'exécution des tâches. Pour cela, nous allons définir $t_{fi} = t_m$ et ce pour tout i. Cela aura des conséquences au niveau des algorithmes de coordination car c'est essentiellement cette modification du système de dates qui fera que les processus ne généreront plus des séquences de dates suivant un flot SPMD.

Pour définir les dates pour les *tâches distantes*, nous avons besoin d'introduire deux nouvelles notions : les *dates complètes* et les *dates restreintes*.

Définition 8 (Date complète)
La date complète d'une tâche t est la date de t considérée dans l'arbre complet d'une simulation, c'est-à-dire l'arbre fusionnant la THG et les THL de la simulation (cf. 2.3.1.1).



(a) $t \in T$ (b) $t \notin T$

FIGURE 2.12 – Répartition des tâches dans la détermination des dates restreintes.

Définition 9 (Date restreinte)
Soit T un arbre de tâche et t une tâche de T, soit T' un sous-arbre de T ayant pour racine t', un ancêtre de t. Considérons $d = d_0.d_1.\cdots.d_{l-1}$ la date de t et $d' = d_0.d_1.\cdots.d_{m-1}$ la date de t' avec $m < l$.
On définit $d|_{T'}$ la date restreinte à T' :
- *si $t \in T'$ alors $d|_{T'} = 0.d_m.\cdots.d_{l-1}$ (voir figure 2.12 (a))*
- *si $t \notin T'$, il existe une tâche t'' tel que t'' soit le premier ancêtre de t appartenant à T' et ayant pour date $d'' = d_0.d_1.\cdots.d_{n-1}$ avec $m < n < l$ alors $d|_{T'} = 0.d_m.\cdots.d_{n-1}$ (voir figure 2.12 (b))*

En règle générale, nous considérerons des tâches restreintes à des *THL* et des *THG*. Il est également à noter qu'une date restreinte à une *THG* est également, par définition, une date complète.

On introduit l'opérateur de concaténation des dates restreintes, que nous noterons $\oplus$. Cet opérateur n'est pas une simple concaténation de dates. En effet, les dates restreintes commencent toutes par un 0, qui est la date de la tâche « *racine* ». Lors de la concaténation de plusieurs dates restreintes, il faut donc retirer ce 0 de toutes les dates sauf la première. Cet opérateur nous permet à partir des dates restreintes d'une tâche d'en calculer la date complète. Par exemple, si nous considérons trois dates restreintes 0.1.0, 0.2.1.1 et 0.1.0.3, la concaténée sera
$0.1.0 \oplus 0.2.1.1 \oplus 0.1.0.3 = 0.1.0.2.1.1.1.0.3$ ce qui correspond au schéma d'addition décalé suivant :

$$
\begin{array}{cl}
 & 0.1.0 \\
\oplus & \quad\;\; 0.2.1.1 \\
\oplus & \qquad\qquad 0.1.0.3 \\
\hline
= & 0.1.0.2.1.1.1.0.3
\end{array}
$$

Pour une *tâche distante*, nous pouvons donc considérer les dates restreintes de la tâche dans la *THG* et les *THLs* ou la date complète. De même, pour une sous-tâche d'une *THL*, nous pourrons considérer soit sa date restreinte à la *THL*, soit sa date complète. De la même façon, on étendra les *dates de point complètes* à partir des dates de tâches complètes.

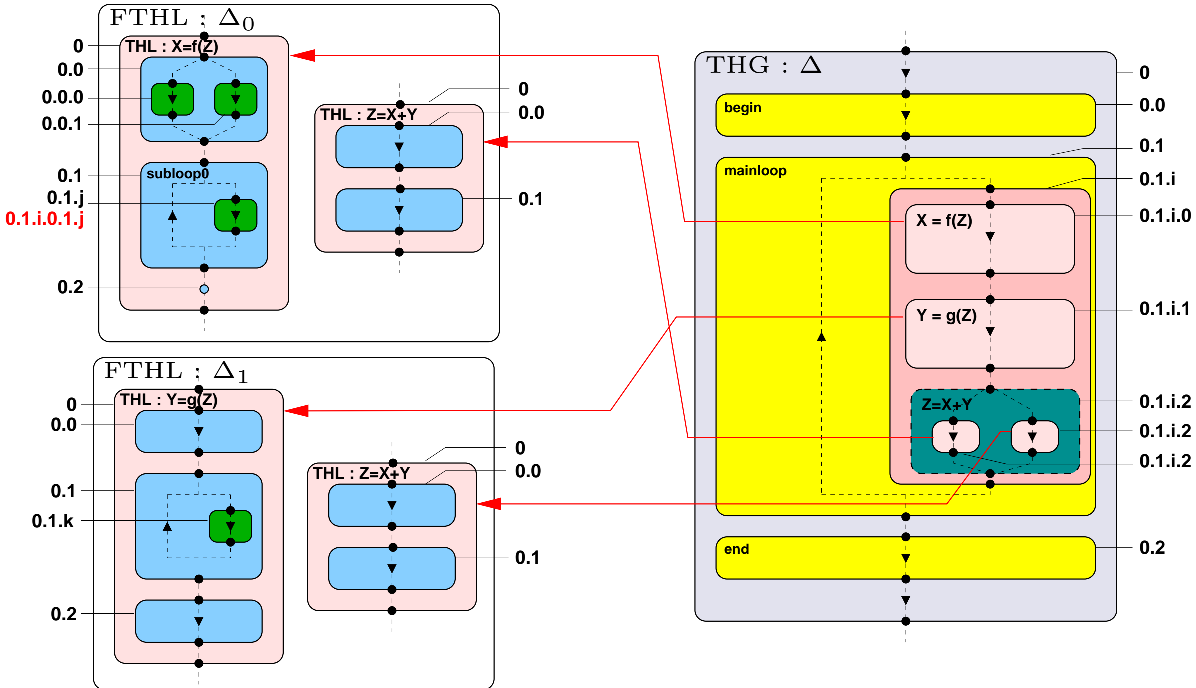


FIGURE 2.13 – Exemple de dates de tâche restreintes et complètes pour la simulation Δ

Afin de clarifier ces nouvelles définitions, nous reprenons notre exemple de la simulation Δ (figure 2.8). Si nous considérons la *THG*, les dates suivent le système de dates du *MHTp*. Ainsi les tâches du premier niveau de la hiérarchie *begin*, *mainloop*, *end* ont pour date 0.0, 0.1 et 0.2. La tâche *mainbody* a pour date $0.1.i$ avec i le nombre d'itérations de la boucle. Les deux sous-tâches suivantes sont des tâches distantes, *X=f(Z)* et *Y=g(Z)* ; elles ont pour date $0.1.i.0$

et 0.1.i.1. Ce sont donc les *dates complètes* pour les *THL X=f(Z)* et *Y=g(Z)*. La sous-tâche suivante est une tâche concurrente contenant deux tâches distantes. Ces deux tâches ont pour *date complète* 0.1.i.2, elles ont donc bien toutes les deux la même date dans la *THG*.

Si nous prenons la description des tâches dans la *FTHL* Δ_0, les dates des tâches restreintes sont construites de la même façon que dans la *THG*. Les dates des tâches de la *THL X=f(Z)* sont donc 0.0 pour la première et 0.1 pour la tâche *subloop0*. La tâche *subbody0* aura pour date 0.1.j, avec j le nombre d'itérations de la boucle. Ces dates sont des *dates locales*.

Une fois les dates restreintes définies dans la *THG* et dans la *THL*, nous pouvons définir les dates complètes des tâches. Si nous reprenons la tâche *subbody0* qui a pour date restreinte à la *THL* 0.1.j et pour date restreinte à la *THG* 0.1.i.0, elle aura pour date complète 0.1.i.0.1.j qui est bien la concaténation de la date 0.1.i.0 et de la date 0.1.j à laquelle on a retiré le premier 0.

Nous avons donc défini un système de dates nous permettant de nous repérer exactement dans le flot d'exécution d'une simulation distribuée représentée dans le *MHTd*, que ce soit dans les parties communes du code, ou bien dans des sous-codes. Par conséquent, nous avons tout ce qu'il faut pour pouvoir définir des algorithmes de planification pour le pilotage des simulations distribuées. Mais pour interagir avec les données de ces simulations, nous avons vu qu'il nous fallait tout d'abord décrire ces données. Or, il nous faut, là aussi, enrichir notre modèle afin de prendre en compte les données des différents codes dans les interactions avec les simulations distribuées.

2.3.1.3 Modèle de description des données

Une fois le flot d'exécution modélisé, nous décrivons les données que nous souhaitons visualiser afin de pouvoir les récupérer de façon cohérente dans le flot d'exécution.

Si nous prenons l'exemple d'une simulation distribuée, les données « *utiles* » pour la visualisation ne sont plus de simples données multi-variées présentes sur un seul code, mais la composition de plusieurs données sur différents codes. Par exemple, la simulation LibMultiScale que nous avons présentée dans la section 1.1.2 est divisée en deux codes ; un code ayant pour données des points et l'autre un maillage. Les points, représentant des atomes, décrivent le matériel à l'échelle atomique, tandis que le maillage le décrit à l'échelle macroscopique. Or, la donnée utile à la visualisation est le ***déplacement***. Au niveau des points, le ***déplacement*** est représenté par les variables de position initiale des points, ainsi que de position actuelle (information contenue dans le support de la donnée). Pour le second code, nous avons directement une variable qui représente le ***déplacement*** et qui est associée au nœud du maillage. Avec la description des données que nous avons introduit précédemment, nous ne pouvons décrire que les particules avec la variable associée (la position initiale), ou que le maillage et la variable associée (le « *déplacement* »). Mais nous n'avons aucun moyen de décrire la donnée ***déplacement*** qui est la composée de ces deux données.

Cette description n'est donc pas suffisante dans le cas distribué car les données que nous souhaitons visualiser sont potentiellement la composition de plusieurs données réparties dans les différents codes. Pour cela nous allons introduire la notion de *méta-donnée* qui est une donnée composée de plusieurs données sur différents codes et qui correspond à une seule donnée au niveau de la visualisation. Une *méta-donnée* est une donnée purement descriptive de haut-niveau qui regroupe plusieurs descriptions de données réelles. Une *méta-donnée* est donc composée comme une donnée classique, d'un *nom* ou identifiant, d'une *liste de variables*, plus des *informations d'accessibilité*. Pour chaque *variable* de la *méta-donnée*, on lui associe l'identifiant de la donnée et de la variable d'origine, ainsi que le nom du code où elle est décrite.

Avec ce nouveau type de donnée, nous devons donc étendre la notion de *contexte* pour pouvoir

définir les *informations d'accessibilité*. En effet, les données visualisables décrites à l'aide des *méta-données* sont composées de plusieurs données distribuées et accédées de manière spécifique sur différents codes et nous devrons donc définir une notion d'accessibilité ad-hoc.

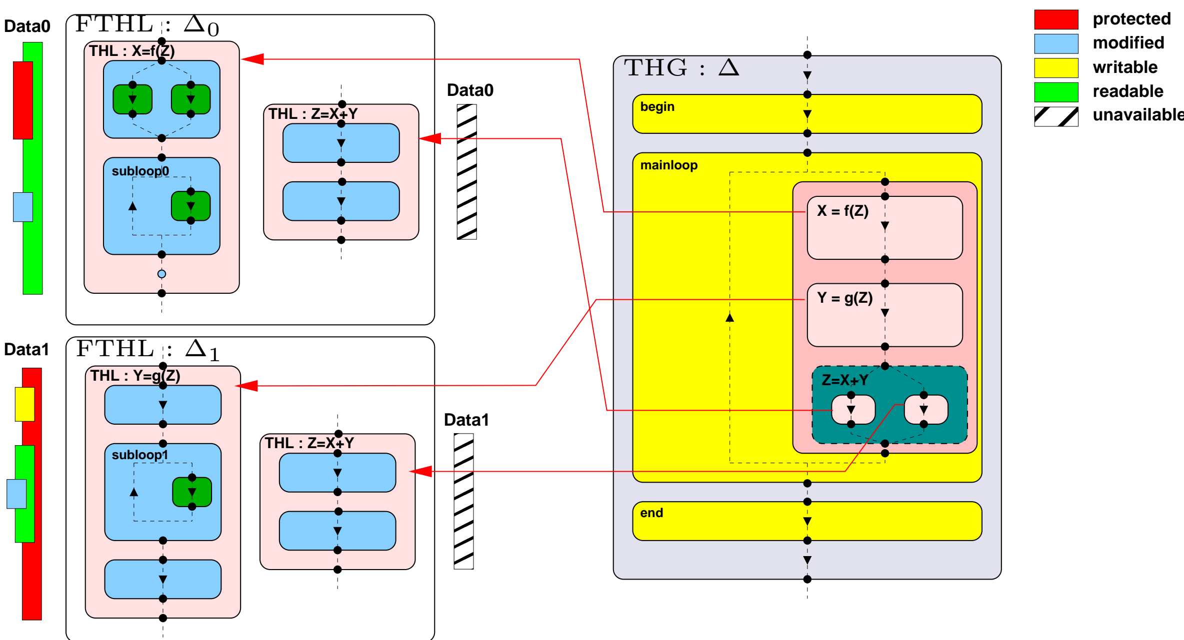


FIGURE 2.14 – Exemple de contexte de donnée appliqué à la description de la simulation Δ

Avant de définir les *contextes d'accès* des *méta-données*, nous avons tout d'abord étendu cette notion aux variables d'une donnée. Les *méta-données* étant constituées d'un ensemble de variables provenant d'autres données, il était trop contraignant d'avoir des *contextes d'accès* pour une donnée complète. De plus les contextes d'accès par variable représentent mieux la réalité des codes de simulation. En effet, toutes les variables d'une donnée ne sont pas modifiées, lisibles ou modifiables en même temps dans un code de simulation. Vu que nous définissons le *contexte d'accès modified* par variable, les *révisions* sont également déterminées par variable. Mais nous pouvons tout de même spécifier les contextes d'une donnée et dans ce cas, ces contextes s'appliqueront à toutes les variables.

Nous définissons donc le *contexte d'accès* d'une *méta-donnée* comme l'union des *contextes d'accès* des variables composant la *méta-donnée*, à l'exception des contextes *modified* de la *méta-donnée* que l'on va restreindre à un sous-ensemble des contextes *modified* des variables. Pour ce faire, nous allons utiliser les *masques de dates* afin de sélectionner les révisions que l'on souhaite accepter pour chaque variable de la *méta-donnée*. Les masques nous permettent de définir les révisions des variables d'une *méta-donnée* qui sont cohérentes entre elles. Les révisions des variables étant les dates des tâches ayant pour contexte *modified*, le fait de se restreindre à un sous-ensemble de l'ensemble de ces révisions, permet de restreindre l'ensemble des tâches avec un contexte *modified* considérées lors du parcours du flot d'exécution. Cette restriction n'est pas forcément nécessaire, mais elle permet par exemple de définir avec précision la version cohérente de la *méta-donnée* lorsque plusieurs révisions des données sont générées au cours d'une même itération, ou lorsque la cohérence n'est pas obtenue pour toutes les itérations des codes. Mais la modification des *informations d'accessibilité* ne se limite pas à cela. En effet, si l'on prend le cas des simulations Client/Serveur, les données peuvent être définies soit pour toutes les

fonctions (*THL*) d'un serveur, soit juste pour certaines fonctions. Par conséquent, la description des données sera associée au serveur (*FTHL*), mais nous ajoutons un contexte permettant de préciser que la donnée n'est pas définie dans une fonction particulière du serveur. Ce contexte sera nommé *unavailable* et sera toujours associé à une *THL* complète.

Afin d'expliquer plus en détails la description des données et des contextes de données, nous allons illustrer notre propos à l'aide de la figure 2.14. Comme précédemment, il s'agit d'une version simplifiée du MHT Δ. Nous allons considérer deux données *Data0* et *Data1* respectivement présentes dans les *THL X=f(Z)* et *THL Y=g(Z)*. *Data0* est lisible par défaut dans toute la *THL X=f(Z)* et une nouvelle version est générée à chaque itération de la boucle *subloop0* dans la tâche *subbody0*. En ce qui concerne la donnée *Data1*, elle est protégée par défaut dans toute la *THL Y=g(Z)* et elle est lisible uniquement lors du passage dans la boucle *subloop1*. De plus, elle est modifiée dans la tâche *subbody1*. Les deux données *Data0* et *Data1* ne sont pas présentes dans les autres *THL* des serveurs et sont donc marquées *unavailable* dans les *THL Z=X+Y*. La *méta-donnée* que l'on va considérer dans cet exemple va être une donnée composée d'une variable de *Data0* et d'une variable de *Data1* pris dans leurs *THL* respectives. Si nous considérons que cette donnée est consistante toutes les 3 itérations de la principale boucle de la *THL X=f(Z)*, tous les 2 tours de la boucle dans la *THL Y=g(Z)* et ceci pour toutes les itérations de la boucle principale *mainloop*, cela nous donne un contexte *modified* pour la *méta-donnée* de type : *modified* avec le masque $0.1.\widetilde{1}.0.1.\widetilde{3}$ pour *Data0* et *modified* avec le masque $0.1.\widetilde{1}.1.1.\widetilde{2}$ pour *Data1*.

Nous venons de présenter comment décrire les données dans le *MHTd*. L'introduction des *méta-données* nous permet non seulement de définir les données utiles à la visualisation, mais également comment accéder à ces données transversales aux différents codes de simulation et de façon cohérente. Il faut tout de même noter que si les contextes de données *modified* sont mal définis par l'utilisateur, la cohérence pourrait ne jamais être obtenue au cours de l'exécution.

2.3.2 Modèle de pilotage

Nous venons de modifier le *MHT*, le système de dates et la description des données ; par conséquent le modèle de pilotage est lui aussi remis en cause par ces nouvelles définitions. Nous devons redéfinir exactement la notion de requête cohérente pour une simulation distribuée et par conséquent nous devons modifier l'algorithme de planification présenté dans la section 2.2.2.2.

2.3.2.1 Pilotage par les requêtes

Dans le cas distribué, la description de la simulation se divise en plusieurs *tâches hiérarchiques*, la *THG* et les tâches contenues dans les *FTHL*. Ces descriptions sont reparties sur les différents processus de la simulation ; ainsi chaque processus ne détient que les informations propres à son code. Contrairement au *MHTp*, la même information n'étant plus présente sur tous les processus, le cycle de vie des requêtes doit être adapté.

Nous devons donc modifier tout le cycle de vie des requêtes afin qu'il nous permette de piloter efficacement des simulations distribuées. La figure 2.15 présente la version modifiée du cycle de vie des requêtes de pilotage sur une simulation distribuée constituée de deux codes. Les étapes 3a, 4a et 5a correspondent aux étapes se déroulant sur le premier code, le *Code A*, et les étapes 3b, 4b et 5b sur le *Code B*. Dans le cas distribué, tout comme dans le cas parallèle, le cycle de vie débute par la réception de la requête au niveau de la simulation (étape 1), suivie d'une étape de coordination (étape 2) nécessaire pour déterminer la date de planification t_{pg} commune aux différents codes. Puis, nous avons une seconde phase de coordination (étapes 3a, 3b) permettant

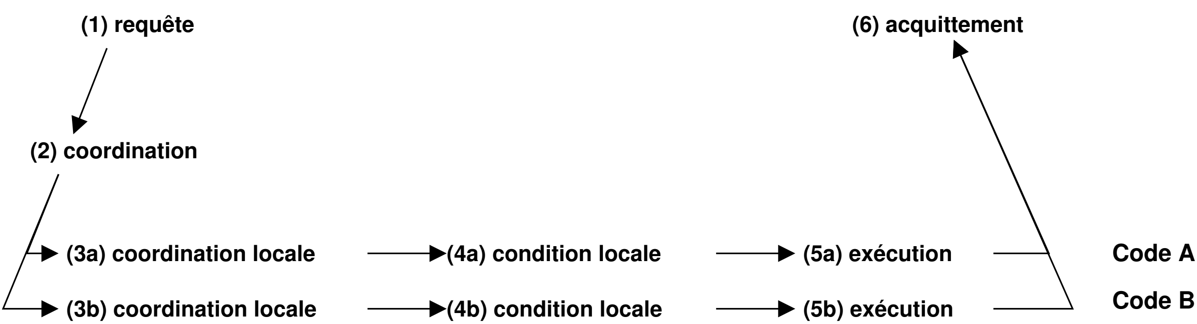


FIGURE 2.15 – Cycle de vie d'une requête de pilotage sur un simulation distribuée composée de deux codes (Code A et Code B)

de raffiner la date de planification en une date t_{pl} restreinte aux *THL* de chaque code. Ensuite, nous vérifions localement les conditions locales aux processus (étapes 4a, 4b). La suite logique est donc l'exécution locale de la requête (étapes 5a, 5b) et la notification de la fin de l'exécution de la requête par l'envoi d'un acquittement global (étape 6).

Notons que nous avons autant de *conditions locales* que de codes dans la simulation distribuée. Comme nous avons modifié le modèle de description des données, cela remet en cause la définition de la cohérence introduite précédemment. Nous devons donc redéfinir la cohérence d'un traitement pour une simulation distribuée. En fait, nous allons définir deux types de cohérence, la *cohérence forte* et la *cohérence faible* ou *partielle*.

Définition 10 (Cohérence forte)
On dira que la cohérence est forte si la même condition locale est satisfaite sur tous les codes d'une simulation distribuée.

La *cohérence forte* correspond donc à une cohérence globale à toute la simulation distribuée, c'est l'équivalent de la cohérence pour des simulations parallèles. Dans la suite lorsque l'on parlera de cohérence sans spécifier le type, il s'agira de *cohérence forte*.

Définition 11 (Cohérence faible)
On dira que la cohérence est faible si une ou plusieurs conditions locales différentes sont satisfaites sur un ou plusieurs codes de la simulation distribuée.

De plus, selon que l'on souhaite obtenir une *cohérence faible* ou une *cohérence forte* pour une même requête, nous aurons des *conditions locales* différentes. Par exemple sur les requêtes de contrôle, la *condition locale* consiste à vérifier que la date planifiée est atteinte. Pour obtenir une *cohérence forte*, cette date sera définie sur une date restreinte au *THG*. Si nous ne voulons qu'une *cohérence faible* la date planifiée de traitement sera une date complète d'une tâche fille d'une *THL* et qui sera potentiellement différente pour chaque code. Nous aurons donc bien des conditions locales différentes suivant le type de cohérence que l'on souhaite obtenir.

Dans le cas des requêtes de données, la *cohérence forte* signifie que les données extraites des codes sont cohérentes entre elles, c'est-à-dire qu'elles correspondent à la description d'une *méta-donnée*. Une *cohérence faible* implique par contre aucun lien entre les différentes données et on cherche uniquement à extraire les données de façon cohérente sur chaque code indépendamment.

Le fait d'avoir deux niveaux de cohérence implique aussi des différences au niveau de la planification. En effet, si l'on veut une *cohérence forte*, il faut déterminer une *date planifiée* sur une date restreinte au *THG* (ou *date planifiée globale*), alors que pour une *cohérence faible*,

il faudra des *dates planifiées* définies sur des dates complètes de sous-tâches de *THL* (ou *dates planifiées locales*). Par conséquent, nous devons définir de nouveaux algorithmes de planification.

2.3.2.2 Les algorithmes de planification

Afin de définir ces nouveaux algorithmes de planification permettant de déterminer des *dates planifiées globales* ou *locales*, nous allons différencier les algorithmes pour les requêtes de contrôle et de données. Nous les différencions car dans les deux cas les besoins ne sont pas exactement les mêmes.

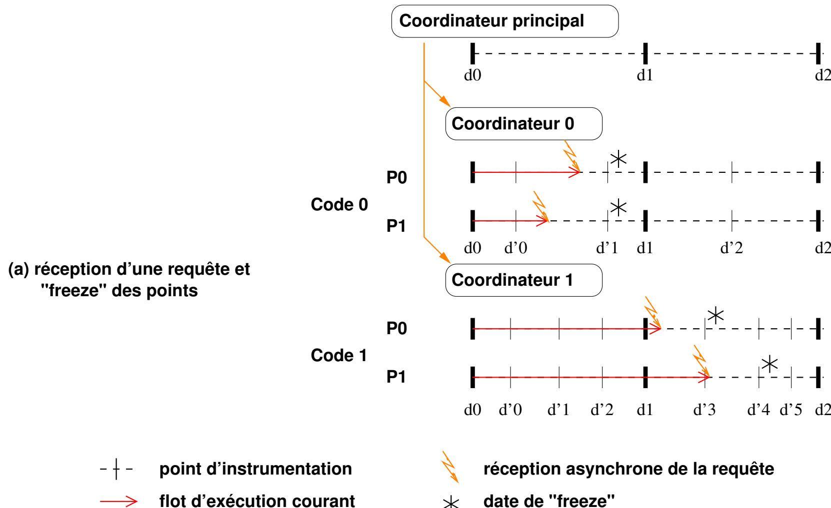
Requête de contrôle – Pour la planification des requêtes de contrôle, nous avons plusieurs solutions, selon le niveau de cohérence que l'on souhaite. Nous avons vu précédemment que la cohérence d'une requête de contrôle est obtenue en vérifiant que la *date planifiée* est atteinte. Par conséquent, en déterminant les dates planifiées locales indépendamment pour chaque code, nous obtenons une cohérence faible. Par contre, si nous planifions une date globale commune à tous les codes, la cohérence sera forte.

Pour les requêtes de contrôle nous avons choisi d'utiliser la cohérence forte. En effet, une *cohérence faible* dans une requête de contrôle permet d'avoir des dates de traitement plus « *fines* », au sens où on peut stopper la simulation à un grain plus fin. En général, il y a plus de dates de tâches locales que de tâches d'une *THG*. Par conséquent, le traitement de requêtes sur des dates planifiées locales sera plus réactif. Le délai entre deux sous-tâches de *THL* étant plus court, le temps d'attente entre la détermination de la date et le traitement sera plus court. Mais, l'inconvénient principal d'une cohérence faible est que les différents codes ne seront pas dans un état cohérent entre eux.

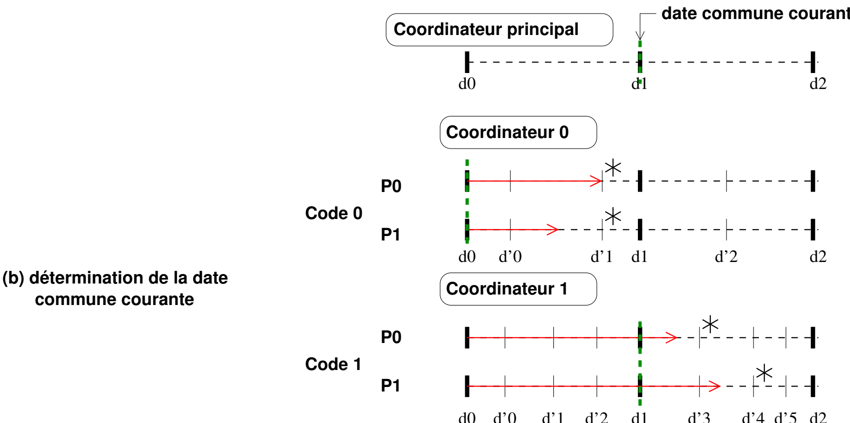
La solution de la *cohérence forte* permet quant à elle de stopper la simulation à un grain plus grossier. Par conséquent, dans le cas d'un appel de méthode à distance non concurrent, nous ne pourrons pas stopper l'exécution dans une sous-tâche d'une *THL* du code appelé, même si la cohérence pourrait être considérée comme forte dans le cas d'appels non concurrentiels. Mais cette solution nous assure la cohérence des différents codes entre eux lors du traitement de la requête.

Nous décrivons maintenant l'algorithme permettant de planifier une date planifiée globale dans une simulation distribuée. Cet algorithme se découpe en trois phases. Sur les différentes illustrations des phases de l'algorithme, les dates de la forme dk sont des dates de tâche restreintes à la *THG* et celles de la forme $d'k$ sont des dates de tâche restreintes aux *THL*.

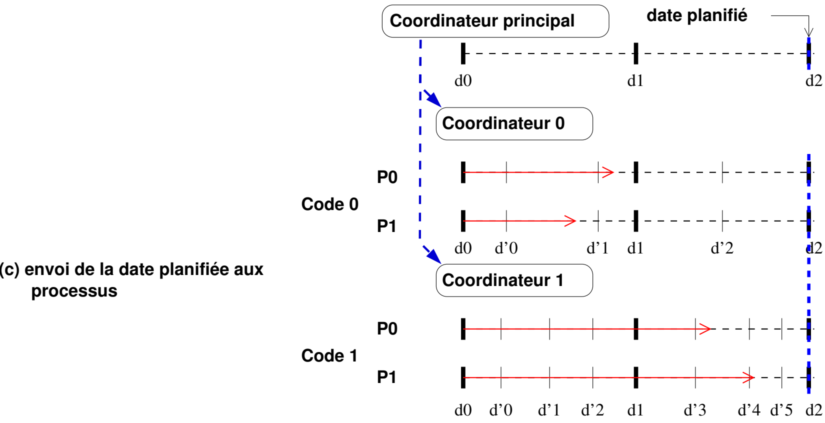
Étape 1 La simulation reçoit la requête via le coordinateur principal qui est un processus permettant de centraliser les requêtes destinées à la simulation. Ce coordinateur envoie un ordre de *freeze* aux coordinateurs associés à chaque code. Il y a donc autant de coordinateurs que de *FTHL*. Sur la figure nous avons deux codes, et par conséquent deux coordinateurs, `Coordinateur 0` et `1`.



Étape 2 Le coordinateur principal détermine la *date commune courante*. Cette date est le maximum des dates courantes restreintes à la *THG* des processus. Pour le `code 0` la date courante est d_0, pour le `code 1` c'est d_1. Le maximum de ces deux dates nous donne donc d_1 comme *date commune courante*.



Étape 3 Le coordinateur en déduit la *date planifiée*, puis l'envoie aux processus concernés. La *date planifiée* est la première date restreinte à la *THG* suivant la *date commune courante*, soit d_2 sur notre exemple. Les processus concernés par la *date planifiée* rendent cette date bloquante. Puis tous les processus relèvent l'ordre de *freeze*.



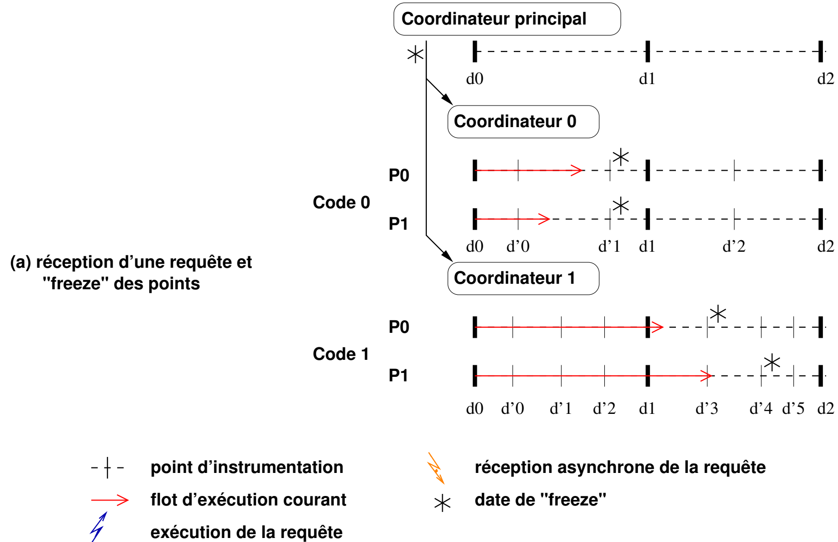
Avec cet algorithme, nous pouvons stopper une simulation distribuée de façon synchronisée dans le flot d'exécution. En ce qui concerne les requêtes de type *play*, il ne s'agit que de retirer

le caractère bloquant des points placés par l'algorithme précédent.

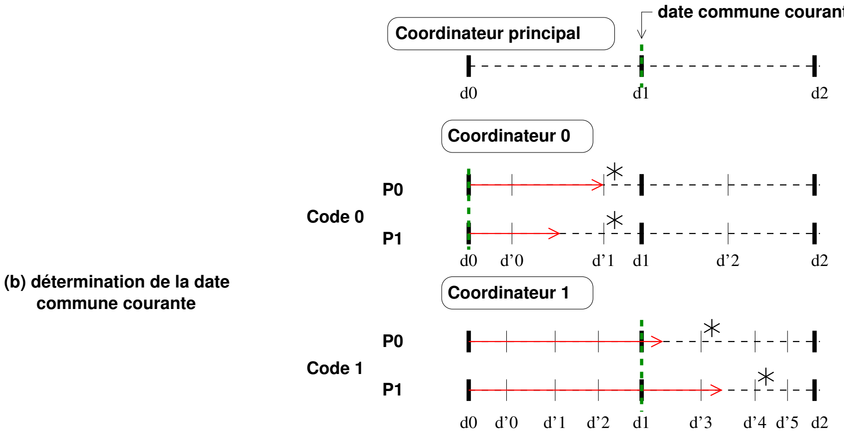
On peut légèrement modifier cet algorithme afin de lever l'inconvénient de la cohérence forte dans le cas d'un appel de méthode à distance non concurrent. Pour ce faire, il suffit de déterminer la date courante commune dans la *THL* concernée. Cela se fait grâce à l'algorithme de planification SPMD (voir section 2.2.2.2) au niveau du coordinateur associé à la *THL*.

Requête de données – Les requêtes d'accès aux données ne peuvent pas être exécutées sur des dates planifiées globales, car nous ne sommes pas sûrs de pouvoir remplir les conditions locales sur de telles dates. Nous n'avons pas besoin d'une cohérence forte du flot d'exécution, mais d'une cohérence forte des données, ce qui n'est pas exactement pareil au niveau de la planification. Tous les codes doivent atteindre la même tâche du *THG*, mais nous pouvons ensuite nous limiter à une planification locale de la date sur les sous-codes de la simulation. L'algorithme est donc un peu plus complexe, il se divise en six phases :

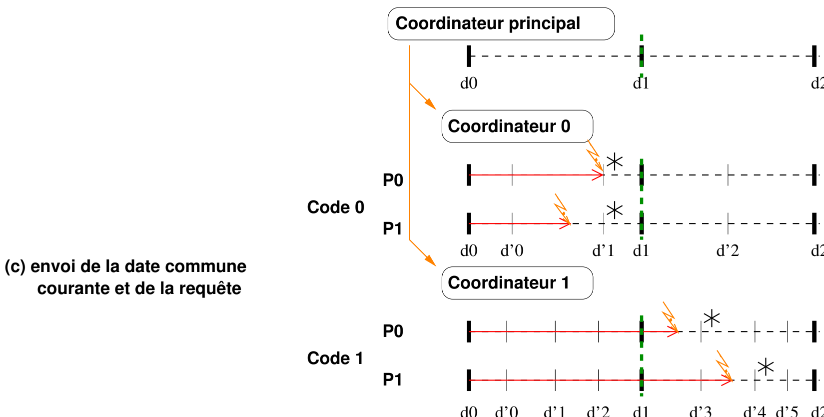
Étape 1 À la réception d'une requête d'accès aux données sur le coordinateur principal, ce dernier envoie un ordre de *freeze* à tous les processus via les coordinateurs locaux.



Étape 2 Le coordinateur principal détermine la *date commune courante* dans le *THG*. Comme précédemment, cette date est le maximum des dates courantes restreintes au *THG* (d_0 et d_1), ici la *date commune courante* est donc d_1.

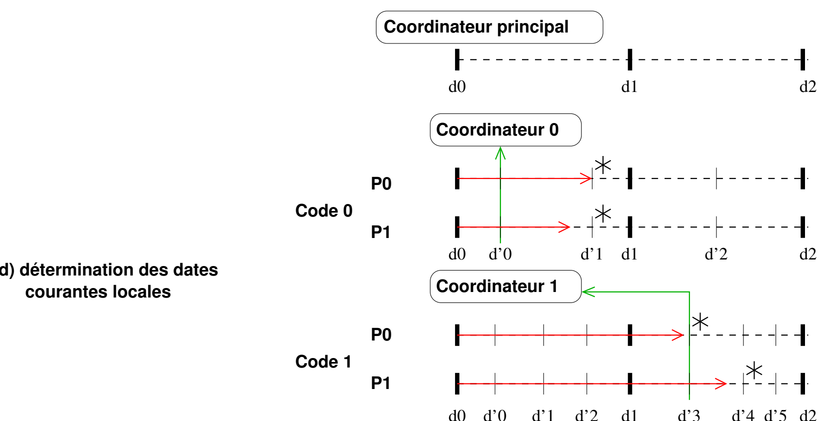


Étape 3 Puis, il envoie la requête et la *date commune courante* à tous les coordinateurs associés à un code détenant une partie de la donnée demandée.



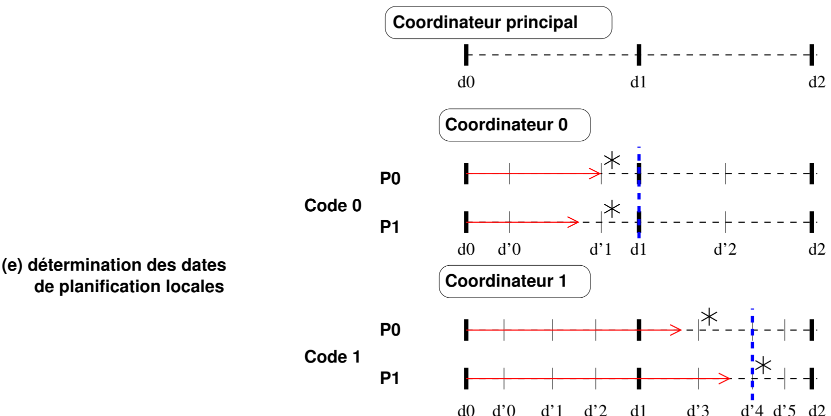
(c) envoi de la date commune courante et de la requête

Étape 4 À la réception de la requête, les coordinateurs locaux déterminent leur *date courante restreinte à la THL* grâce à l'algorithme de planification SPMD : d'_0 sur le `code 0` et d'_3 sur le `code 1`.



(d) détermination des dates courantes locales

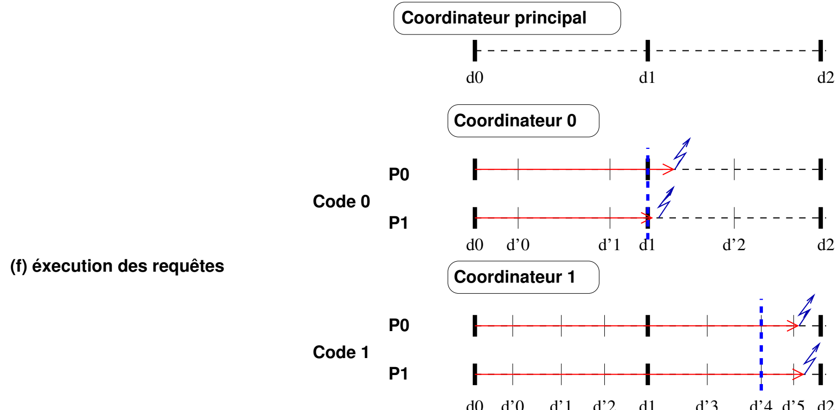
Étape 5 Une fois la *date courante restreinte à la THL* récupérée, les coordinateurs locaux déterminent leur *date planifiée locale* en prenant soit la date qui suit la date qui est la concaténée des deux dates précédemment déterminées, soit la date commune courante dans le cas ou elle n'est pas encore atteinte. Sur le schéma cela nous donne les dates d_1 pour le `code 0` et la date qui suit $d_1 \oplus d'_3$ pour le `code 1` soit $d_1 \oplus d'_4$.



(e) détermination des dates de planification locales

Étape 6 La requête est exécutée sur les processus lorsque la *date planifiée* est atteinte et que les conditions locales décrites par les contextes des méta-données sont satisfaites. Suivant les conditions locales, la date réelle de l'exécution peut être bien après la date planifiée. Cela dépend des plages d'accès et des révisions des données. En effet, il faudra attendre la

première tâche où la donnée a une révision correspondant au masque de date et ayant les droits d'accès nécessaire.



(f) éxecution des requêtes

Nous avons défini un algorithme de planification pour les requêtes de données. Avec cet algorithme, nous pouvons planifier des dates de traitement pour tous les types de requêtes nécessaires au pilotage des simulations distribuées et exécuter ces requêtes de façon cohérente.

2.3.2.3 Cohérence des données dans les clients de pilotage

Nous avons donc tous les outils nécessaires pour assurer la cohérence des traitements effectués sur les simulations distribuées. Il ne reste donc plus qu'à être sûr de ne pas introduire d'incohérence au niveau des clients de pilotage et ce qu'ils soient séquentiels ou parallèles. Au niveau des clients, nous avons deux sources potentielles d'incohérences : les requêtes permanentes et les clients parallèles. Les requêtes permanentes posent le problème du chevauchement potentiel des requêtes dans le client, alors que les clients parallèles posent le problème des données distribuées entre les nœuds des clients.

Ces incohérences peuvent intervenir durant le processus de traitement des données dans les clients de pilotage. Ainsi il nous faut être sûr que la cohérence des données n'est pas remise en cause entre le moment où un client commence à les recevoir et le moment où il a fini de traiter les données. Les traitements de données d'un client de pilotage consistent essentiellement en des transformations pour pouvoir facilement interpréter ces données. Nous avons vu que dans le cycle de vie d'une requête, la fin des traitements est notifiée par une phase d'acquittement. Le client a plusieurs possibilités pour l'acquittement : il peut soit *acquitter au plus tôt*, soit *acquitter au plus tard* (voir figure 2.16). L'*acquittement au plus tôt* consiste à envoyer l'acquittement à la simulation dès que la réception des données est terminée. L'*acquittement au plus tard* consiste à attendre la fin de tous les traitements avant d'acquitter la simulation.

L'*acquittement au plus tôt* (figure 2.16a) permet de rendre la main le plus tôt possible à la simulation et donc de réduire le surcoût potentiel engendré sur le temps d'exécution de la simulation. En effet, comme nous l'avons vu précédemment, si la condition locale d'une requête devait ne plus être vérifiée, le flot d'exécution de la simulation serait alors suspendu jusqu'à la fin des traitements de la requête. Ainsi, le fait d'acquitter au plus tôt permet de notifier la fin de la requête plus tôt et donc de débloquer au plus tôt la simulation suspendue. Dans le cas de requêtes de données non permanentes, cela est suffisant car ce sont des requêtes engendrées par un utilisateur, et souvent l'utilisateur ne redemande pas une donnée avant d'avoir vu le résultat de la requête courante. Il n'y a donc, a priori, pas de chevauchement de requêtes, c'est-à-dire qu'aucune requête ne débute alors que la précédente n'est pas finie. Dans le cas de requêtes permanentes, ce type d'acquittement peut mener à des incohérences de données. En effet, dans

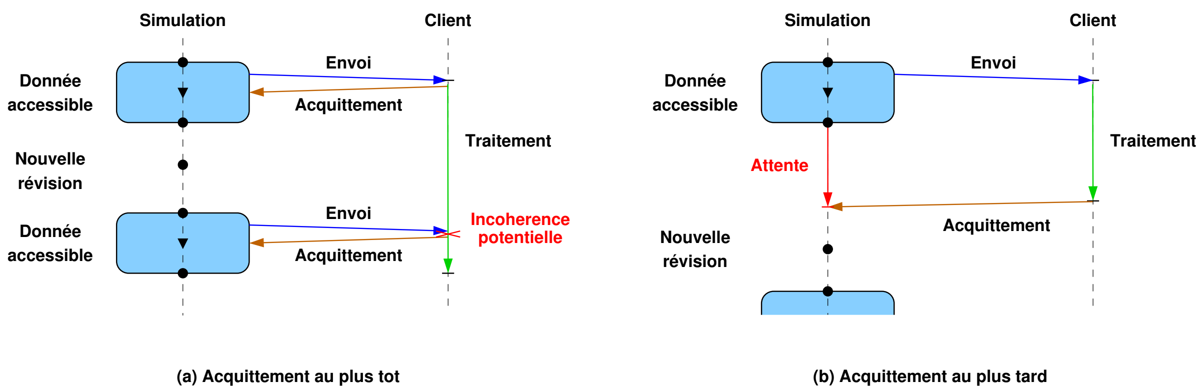


FIGURE 2.16 – Stratégies d'acquittement au plus tôt (a) et au plus tard (b)

ce cas, les données sont automatiquement envoyées à chaque nouvelle révision. Or, si la donnée est envoyée à un client qui n'a pas encore fini ses post-traitements, la donnée dans la mémoire du client peut être modifiée pendant qu'il l'utilise. On arriverait à un cas où la donnée est incohérente en mémoire (*i.e.*plusieurs révisions entrelacées).

Pour éviter cette situation, nous pouvons acquitter la donnée au plus tard (figure 2.16b). Ainsi, nous ne pouvons pas générer une nouvelle révision de la donnée au niveau de la simulation avant que les traitements du client ne soient totalement terminés. Cela nous assure la cohérence des données dans le client, mais cette méthode peut engendrer d'importants surcoûts sur le temps d'exécution de la simulation.

Dans le cas de clients parallèles, un autre problème de cohérence intervient car les données peuvent arriver sur les différents nœuds de façon asynchrone. Dans ce cas, les données distribuées entre les différents nœuds peuvent être dans un état incohérent. Il faut donc s'assurer de ne débuter les traitements dans le client que lorsque la réception des données est terminée. Pour cela, on peut utiliser un coordinateur au niveau du client qui est acquitté à la fin de la réception des données par les différents processus des clients. Ce coordinateur peut donc commander les post-traitements après s'être assuré que les données étaient bien réceptionnées sur tous les nœuds. Le coordinateur peut donc également choisir d'acquitter au plus tôt ou au plus tard.

La méthode de l'acquittement au plus tard est potentiellement coûteuse pour la simulation. Nous avons donc introduit une autre méthode permettant d'assurer la cohérence dans les clients parallèles. C'est une méthode qui permet d'acquitter au plus tôt et donc de potentiellement augmenter le recouvrement des post-traitements par des tâches de la simulation. Cette méthode consiste à bloquer l'accès au données pendant les post-traitements. Ainsi, une nouvelle version de la donnée ne peut pas être réceptionnée tant que des traitements sont en cours. La difficulté réside dans le blocage de la bonne révision pour tous les processus. Cela revient à faire un « *lock* » parallèle pour une certaine révision d'une donnée. Pour ce faire nous avons un algorithme de *lock global* en cinq étapes inspiré de l'algorithme de planification :

Étape 1 Le *coordinateur* demande aux processus de prendre le *verrou* sur la variable d'une donnée.

Étape 2 Les processus prennent le *verrou* dès que possible et envoient la révision courante de la donnée qu'ils viennent de verrouiller au *coordinateur*.

Étape 3 Le *coordinateur* détermine la révision courante globale en prenant le maximum des révisions courantes des différents processus. Puis il envoie cette révision à tous les processus.

Étape 4 Les processus verrouillent la révision courante globale. Pour ce faire, soit ils ont déjà verrouillé la bonne révision et ils ne font rien, soit ils n'ont pas verrouillé la bonne révision

et ils libèrent le *verrou* pour le reprendre lorsque la révision sera atteinte. Dans les deux cas, ils notifient le *coordinateur* lorsque la révision est verrouillée.

Étape 5 Le *coordinateur*, une fois notifié par tous les nœuds, considère que le *lock global* est pris et force les processus à débuter leurs traitements sur les données tout en acquittant la simulation.

Avec l'*acquittement au plus tard* et l'algorithme de *lock global* nous avons donc deux manières d'assurer la cohérence des données dans les clients de pilotage, qu'ils soient séquentiels ou parallèles. Ainsi, nous pouvons garantir la cohérence des données sur toute la chaîne de traitement, depuis la récupération dans la mémoire des simulations couplées jusqu'à la fin des traitements de visualisation.

2.4 Conclusion

Dans ce chapitre, nous avons présenté un modèle générique permettant de modéliser des simulations distribuées de types M-SPMD ou Client/Serveur. Ce modèle offre une vision simple et structurée de ces simulations. De plus, le système de datation s'appuyant sur ce modèle permet de se repérer précisément dans le flot d'exécution. Ce modèle est une extension du modèle présenté dans [40] dont nous conservons les propriétés sur les dates. Nous pouvons les ordonner et donc planifier des actions de façon cohérente en temps sans avoir à synchroniser fortement les processus. L'algorithme de planification a été étendu afin de pouvoir planifier des actions sur des codes distribués, mais surtout pour assurer la cohérence des données distribuées. Cela a été rendu possible grâce à une description plus fine des données distribuées via l'introduction des *méta-données* et des masques de dates.

Chapitre 3

Réalisation d'une plate-forme de pilotage pour les simulations numériques distribuées

Sommaire

3.1 Introduction **72**
3.2 La plate-forme EPSN2 . . . . . . **72**
3.2.1 Vue d'ensemble 73
3.2.1.1 Extension du noyau de EPSN 74
3.2.1.2 ColCOWS 75
3.2.1.3 RedGRID 77
3.2.2 Intégration d'une simulation distribuée dans EPSN 77
3.2.3 Implantation des différents algorithmes et schémas de communication . 79
3.2.3.1 Algorithme de planification des requêtes de contrôle 80
3.2.3.2 Algorithme de planification des requêtes de données 82
3.2.3.3 Difficultés des simulations Client/Serveur 83
3.2.3.4 Extensions des schémas de communication 84
3.3 Client de pilotage et de visualisation **86**
3.3.1 Clients EPSN 86
3.3.1.1 Briques de base des clients EPSN 87
3.3.1.2 Clients séquentiels 88
3.3.1.3 Clients parallèles 90
3.4 Conclusion **91**

Dans le chapitre précédent, nous avons présenté notre modèle pour le pilotage de simulations distribuées de type M-SPMD ou Client/Serveur, ainsi que les algorithmes associés. Nous allons maintenant décrire l'implantation de ce modèle dans la plate-forme de pilotage EPSN.

3.1 Introduction

EPSN est un environnement de pilotage de simulations parallèles, nous allons l'étendre afin de prendre en compte les simulations couplées. Pour rester le plus générique possible dans le cadre du pilotage de simulations distribuées, nous avons tout d'abord étendu le modèle de représentation des simulations, le MHT (voir chapitre 2). Ce modèle permet à EPSN de se repérer dans le flot d'exécution afin d'assurer la cohérence des traitements. Il permet également de définir des plages d'accès afin de limiter l'impact d'EPSN sur le temps d'exécution de la simulation lors des transferts de données. Les extensions que nous allons apporter à la plate-forme EPSN doivent conserver ces propriétés.

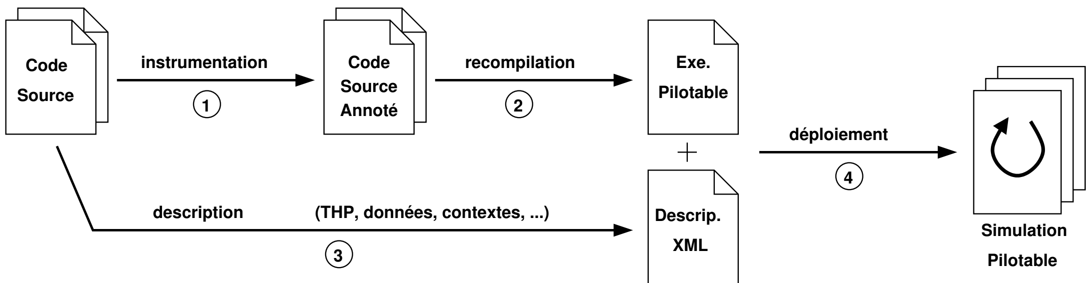


FIGURE 3.1 – Comment rendre une simulations pilotable

Pour piloter une simulation avec EPSN, il faut suivre quatre étapes. La figure 3.1 détaille le processus permettant de rendre une simulation pilotable. Nous commençons par instrumenter le code source (1) à l'aide de l'API simulation d'EPSN ou *back-end API*. Puis nous recompilons ce code afin d'avoir un fichier exécutable de la simulation (2). En plus de l'instrumentation du code, il faut décrire d'une part le flot d'exécution sous forme de tâches hiérarchiques (THP), et d'autre part les données et les contextes d'accès à ces dernières (3). Cette description est effectuée à l'aide du langage XML. Une fois ces trois étapes terminées, nous pouvons déployer la simulation (4). Le déploiement de la simulation se fait de la même façon que si elle n'était pas instrumentée. Lorsque la simulation est lancée, on peut s'y connecter avec n'importe quel client de pilotage et/ou de visualisation construit à partir de l'API cliente d'EPSN ou *front-end API*. Ces quatre étapes nous permettent de piloter une simulation parallèle. Nous verrons dans la section 3.2.2, comment intégrer une simulation distribuée dans EPSN2, mais l'idée principale restera la même, à savoir instrumenter le code et décrire la simulation en XML avant de déployer les codes.

3.2 La plate-forme EPSN2

Dans la section 1.4.2, nous avons déjà présenté dans les grandes lignes la plate-forme EPSN version 1. Nous détaillons maintenant les extensions apportées dans la version 2. Nous décrivons tout d'abord l'architecture générale de EPSN2. Puis, nous expliquerons comment instrumenter et décrire une simulation distribuée. Enfin, nous détaillerons l'implantation effective des algo-

rithmes de planification, ainsi que les différents schémas de communication permettant d'initialiser la plate-forme et de connecter des clients.

3.2.1 Vue d'ensemble

La plate-forme EPSN est basée sur différentes couches logicielles : le noyau de EPSN, ColCOWS et RedGRID. Le noyau de EPSN (EPSN Core) contient toute la partie algorithmique et gère les communications internes à la plate-forme : il gère l'initialisation de la plate-forme, les connexions/déconnexions des différents clients ainsi que les requêtes de pilotage de ses clients. Il implante également les API back-end et front-end pour respectivement instrumenter les simulations et développer les clients. RedGRID (voir section 1.3.2) est la bibliothèque permettant d'échanger les données entre la simulation et les différents clients en proposant des algorithmes de redistribution pour les différents types de données considérés (grilles, points, *etc.*). Enfin, ColCOWS est la bibliothèque permettant d'une part le partage de références des objets CORBA au sein d'un espace de travail, et d'autre part la connexion et la déconnexion d'espace de travail entre eux. Les interfaces entre ces couches logicielles sont représentées sur le schéma 3.2.

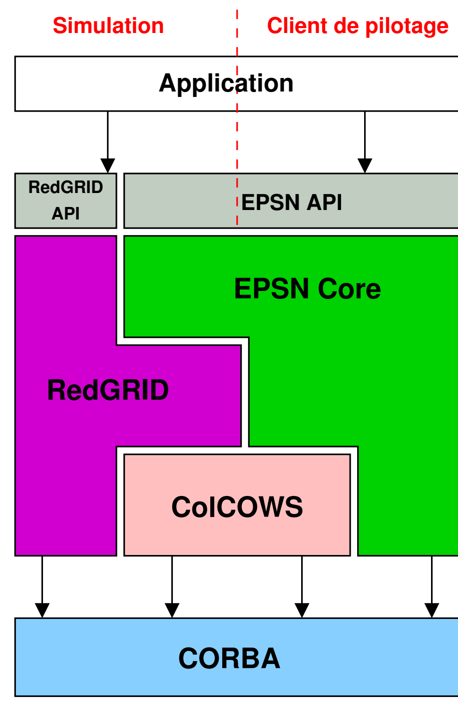


FIGURE 3.2 – Couches logicielles de EPSN

La plate-forme EPSN s'appuie sur l'intergicel de communication CORBA, que ce soit pour les communications internes à la plate-forme, mais également pour les communications entre les simulations et les clients. Cette bibliothèque a été choisie pour la dynamicité et le support de plate-forme hétérogène qu'elle offre. En effet, CORBA nous permet non seulement de pouvoir connecter et déconnecter les clients de pilotage des simulations à volonté, mais également de pouvoir exécuter les simulations et les clients sur des plates-formes différentes sans trop de difficultés, et ce même s'il n'existe à l'heure actuelle que peu d'implantations performantes de la spécification CORBA pour les clusters HPC. Sur ce dernier point, les points critiques sont essentiellement la copie des arguments lors des appels de méthodes ainsi que les appels de

fonctions empilées.

Les extensions du modèle de description ainsi que les algorithmes de planification impliquent essentiellement des modifications du noyau EPSN. La couche RedGRID traite de la redistribution de données ainsi que de la description bas niveau de ces dernières. Les extensions du modèle de description des données de EPSN sont de haut niveau, et par conséquent la bibliothèque RedGRID n'est pas touchée directement par ces modifications. La bibliothèque ColCOWS n'est pas du tout concernée par la modification de la modélisation, mais elle a été modifiée afin d'améliorer ses performances. Nous allons maintenant détailler les modifications apportées à la plate-forme EPSN.

3.2.1.1 Extension du noyau de EPSN

La plate-forme EPSN est basée sur la notion de couplage Client/Serveur entre l'interface utilisateur de pilotage/visualisation (client) et la simulation (serveur). Les applications, simulation ou client, sont vues comme des objets CORBA parallèles, c'est-à-dire un ensemble d'objets identiques distribués sur les nœuds. À chaque nœud d'une simulation parallèle est associé un objet CORBA appelé nœud EPSN (voir figure 3.3), et les nœuds d'une même simulation sont regroupés dans un même espace de travail (*workspace*). Ces espaces de travail sont gérés par la bibliothèque ColCOWS dont on précisera le fonctionnement plus en détails par la suite. La simulation est donc vue comme une seule entité avec un point d'accès unique, appelé *proxy* EPSN. Ce proxy est souvent le processus de rang 0 de la simulation, mais il peut s'exécuter dans un processus complètement séparé des processus de la simulation. Ce *proxy* détient la description haut niveau (XML) de la simulation et la fournit à tous les nœuds de la simulation ainsi qu'aux différents clients lorsqu'ils se connectent.

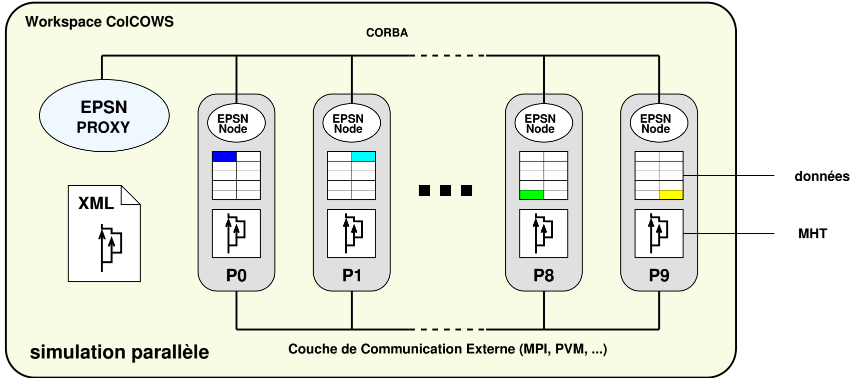


FIGURE 3.3 – Architecture de EPSN1 pour une simulation parallèle

Comme le montre la figure 3.4 dans le cas d'une simulation distribuée, nous n'aurons pas un unique espace de travail ColCOWS par simulation couplée, mais plutôt un espace de travail par code dans la simulation et donc un proxy par code. Ce choix d'architecture a été fait pour de multiples raisons. Tout d'abord, en gardant un espace de travail par code de simulation, cela permet de continuer à voir les codes de simulation comme des entités à part entière. De plus pour chaque code, nous gardons les descriptions des données et du flot d'exécution localement sur ce code. Finalement, les clients peuvent se connecter normalement à la simulation couplée mais aussi se connecter à un seul code en ignorant le reste de la simulation couplée. Pour maintenir la vision de la simulation comme une seule entité, nous devons introduire un point d'accès unique pour la simulation. Ce point d'accès est appelé le *master proxy*. Il partage un même espace de

travail avec les *proxies locaux* à chaque code comme le montre la figure 3.4 et il permet aux *proxies* de s'échanger les informations de haut-niveau sur la simulation couplée et ses différents codes.

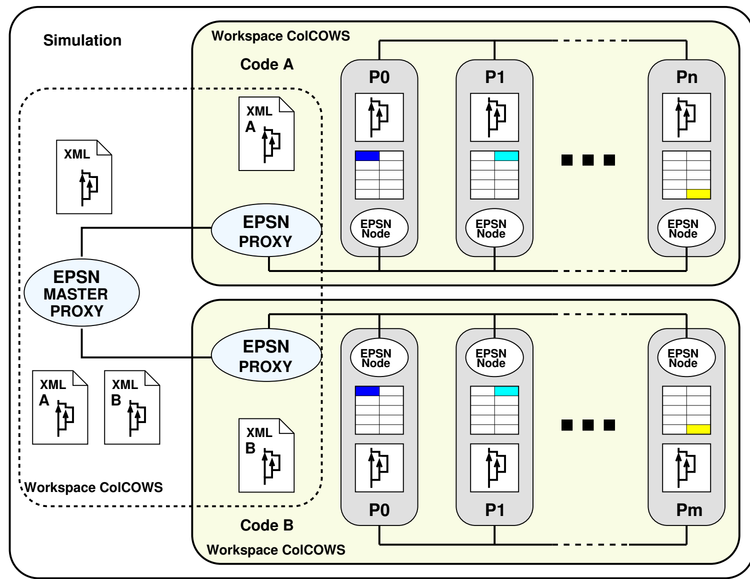


FIGURE 3.4 – Architecture de EPSN2 pour une simulation distribuée

Le *master proxy* gère les connexions des clients ainsi que les différentes requêtes de pilotage. Lorsqu'il reçoit une requête, il l'envoie aux codes concernés. C'est également lui qui interprète les différentes descriptions XML de la simulation couplée et des codes associées et les envoie aux codes concernés. Il intervient également dans le processus de planification car il est le seul à avoir une vision globale de la simulation couplée. Ce choix d'architecture permet de rester compatible avec la version 1 de EPSN, et autorise les clients à se connecter directement aux codes d'une simulation distribuée via les *proxies locaux* ; on conserve ainsi la vision SPMD des codes d'une simulation distribuée.

3.2.1.2 ColCOWS

ColCOWS (Collective CORBA Object WorkSpace) est une bibliothèque développée au sein du projet EPSN permettant la mise en commun d'objets CORBA dans un même espace de travail ou *workspace*. Un objet CORBA est identifié par une référence unique (IOR) à partir de laquelle il est possible de retrouver l'objet sur un réseau et de s'y connecter pour invoquer des méthodes à distance. Dans ColCOWS, la mise en commun se fait par un échange des références des objets détenus par les nœuds (*ColCOWS_Node*) d'un même *workspace*. Chaque *workspace* est identifié par une chaîne de caractère (*WS_ID*) et chaque nœud est identifié par son rang. Une fois associé à un *workspace*, les nœuds peuvent déclarer des objets de même type identifiés par une chaîne de caractère (*Object_ID*). La mise en commun de tous les objets d'un *workspace* se fait durant la phase d'activation (voir figure 3.5), phase pendant laquelle tous les nœuds prennent connaissance de l'existence des autres nœuds. Cela se fait en s'appuyant sur le service de nommage de CORBA. Le nœud 0 inscrit *WS_ID* dans le service de nommage, puis tous les autres nœuds récupèrent l'IOR du nœud 0 dans ce service de nommage. Ensuite, ils communiquent tous leurs IOR au nœud 0 qui se charge de rassembler toutes les IOR avant de les diffuser à tous les nœuds. Par

la suite, tous les nœuds d'un *workspace* peuvent accéder à n'importe quel objet avec la simple connaissance de son *Object_ID* et du rang du nœud qui l'héberge.

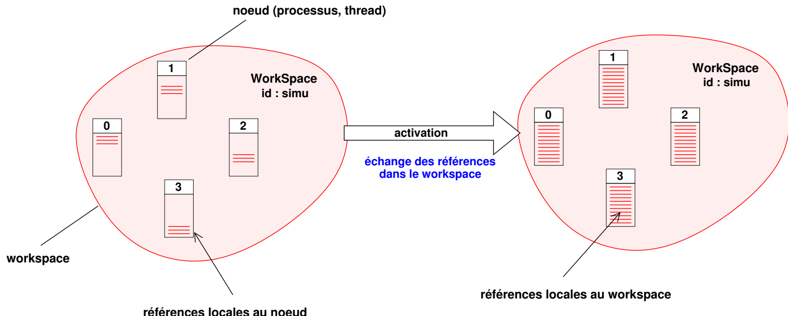


FIGURE 3.5 – Principe de fonctionnement de ColCOWS : activation

Mais ColCOWS ne se limite pas à la mise en commun de références au sein d'un *workspace*. Une fois activés, les *workspaces* peuvent être connectés entre eux pour permettre aux nœuds des deux *workspaces* de prendre connaissance des références des objets de l'autre *workspace*. Cette étape est présentée par la figure 3.6. Lors de la connexion de deux *workspaces*, les nœuds 0 s'interconnectent et s'échangent toutes les références. Une fois que les nœuds 0 ont obtenu les informations, ils les diffusent aux autres nœuds de leur *workspace*.

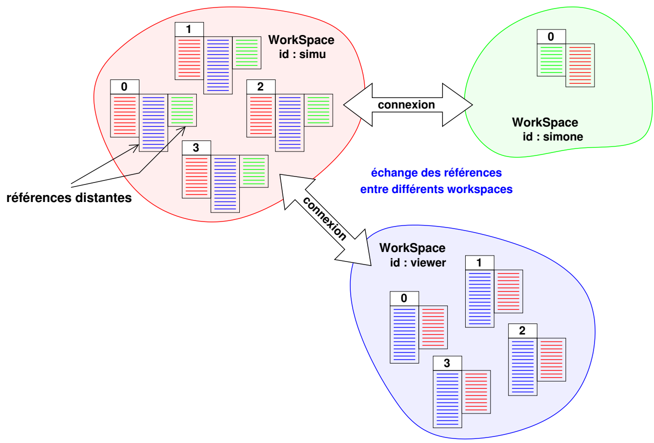


FIGURE 3.6 – Principe de fonctionnement de ColCOWS : connexion

Afin de pouvoir effectuer les étapes d'activation et de connexion efficacement, on utilise des algorithmes de communication collective (*broadcast*, *gather*, *reduce*, *etc.*). Les appels de méthodes CORBA peuvent s'apparenter à des communications point à point, mais CORBA ne propose pas de communication collective « *à la MPI* ». Ces algorithmes ont donc été implantés dans ColCOWS à partir d'appels de méthodes effectués d'après un arbre binaire recouvrant les nœuds. Ces algorithmes sont inspirés des algorithmes que l'on peut trouver dans certaines implantations de MPI, et pour améliorer ces communications collectives, nous utilisons des arbres de communication. La bibliothèque ColCOWS utilise non seulement ces communications collectives en interne lors de l'activation et de la connexion, mais permet également aux nœuds d'utiliser ces types de communication via l'API de la bibliothèque. Elles sont par contre limitées car elles permettent uniquement de diffuser des données à partir du *ColCOWS_Node* 0 vers les autres (*broadcast*), ou de rassembler des données des différents nœuds vers le *ColCOWS_Node*

0 (*gather*, *reduce*). Afin d'être le plus générique possible, ces communications sont basées sur des transferts de données de type ***Any*** de CORBA. Ce type de données permet d'encapsuler n'importe quel type de données.

3.2.1.3 RedGRID

RedGRID [41] est la bibliothèque développée au sein du projet EPSN pour redistribuer des données entre des simulations parallèles d'une part, et des clients de pilotage potentiellement parallèles d'autre part. Cette bibliothèque, comme nous l'avons déjà vu dans la section 1.3.2, permet de décrire la distribution des données sur les codes ainsi que l'agencement en mémoire de ces données grâce à une API nommée RedSYM. Une fois décrites, les données peuvent être transférées d'un code parallèle à un autre via la bibliothèque RedCORBA qui s'appuie sur CORBA pour effectuer les communications.

Dans le cadre du pilotage et surtout de la visualisation des données, nous allons utiliser ces outils. La bibliothèque fournit un ensemble d'objets C++ permettant d'intégrer des données décrites à l'aide de RedSYM directement dans un pipeline de visualisation VTK [141]. Ces classes sont regroupées dans la bibliothèque RedSYM2VTK. Une autre bibliothèque permet de stocker les données au format HDF5 [60] ; il s'agit également d'un ensemble de classes permettant de passer de la description de données RedSYM à une description HDF5. Cette bibliothèque se nomme RedSYM2HDF5. Nous verrons dans la suite de cette section comment EPSN utilise RedSYM2VTK pour visualiser les données dans les clients de pilotage.

3.2.2 Intégration d'une simulation distribuée dans EPSN

Nous allons maintenant présenter les différentes étapes pour intégrer une simulation distribuée dans la plate-forme EPSN. Dans le cas d'une simulation SPMD, il faut instrumenter le code source de la simulation et décrire le flot d'exécution, les données et les contextes d'accès. Dans le cas de simulations distribuées, il faut suivre les même étapes d'instrumentation et de description pour chacun des codes de la simulation. Mais nous avons une phase supplémentaire de description du couplage qui correspond à la construction de la tâche hiérarchique globale (THG) et des méta-données.

Ces fichiers XML sont découpés en autant de morceaux qu'il y a de codes de simulation. Pour chaque simulation, nous avons une description des données et une description des tâches hiérarchiques qui incluent pour chaque donnée les contextes d'accès. Afin d'illustrer notre propos, nous allons décrire en XML la simulation Δ présentée dans la section 2.3.1. Le listing 3.1 correspond à une description XML partielle de cette simulation où nous n'avons gardé que la partie la plus pertinente. La description complète se trouve dans les annexes A.2 et A.3. Pour décrire les simulations distribuées, il a fallu étendre les possibilités du langage de description de simulation de EPSN en ajoutant les balises suivantes :

- ***<simulation>*** les sections ***simulation*** permettent de définir les descriptions des différents codes. Les codes sont différenciés par leur ***id***. Parmi ces codes, l'un d'entre eux joue un rôle particulier, il représente la simulation dans son intégralité. Dans notre exemple, c'est la simulation dont l'***id*** vaut *"delta"* et qui détient la *THG*.
- ***<meta-object>*** cette section permet de décrire une méta-donnée telle que définie dans les chapitres précédents. Il s'agit de composer plusieurs données dans différents codes pour en faire une seule. Si l'on souhaite définir une cohérence simple (la cohérence est obtenue à chaque nouvelle révision de la donnée), on peut omettre l'attribut ***mask***. L'attribut ***mask*** permet de définir le masque de date servant à sélectionner avec précision les révisions pour

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<epsn>
  <simulation id="delta">
    <data>
      <meta-object id="values">
        <meta-variable id="values-0" data_id="Data0"
                       variable_id="values"
                       task_id="procedure00"
                       simulation_id="delta0"
                       mask="0.1.1t.0.1.3t" />
        <!-- ... -->
      </meta-object>
    </data>

    <ght auto-start="false">
      <task id="begin" />

      <loop id="mainloop">
        <task id="mainbody">
          <remote id="remote0" task_id="procedure00"
                  simulation_id="delta1" />
          <remote id="remote1" task-id="procedure10"
                  simulation_id="delta2" />

          <concurrent id="concurrent">
            <remote id="remote0" task_id="procedure01"
                    simulation_id="delta1" />
            <remote id="remote1" task_id="procedure11"
                    simulation_id="delta2" />
          </concurrent>
        </task>
      </loop>

      <task id="end" />
    </ght>
  </simulation>

  <simulation id="delta1">
    <data>
      <!-- ... -->
    </data>

    <lht id="procedure00">
      <switch id="switch">
        <task id="switch0" />
        <task id="switch1" />
      </switch>

      <loop id="subloop0">
        <task id="subbody0"/>
      </loop>

      <point id="point" />
    </lht>

    <lht id="procedure01">
      <!-- ... -->
    </lht>
  </simulation>


  <simulation id="delta2">
    <!-- ... -->
  </simulation>
</epsn>
```

Listing 3.1 – Description XML correspondant a la simulation Δ

chaque donnée formant la méta-donnée cohérente. Les descriptions complètes des ***meta-object*** de la simulation Δ sont disponibles dans le listing A.2. Dans les listings en annexe, on peut également voir les contextes des données dans les codes serveurs $\Delta 1$ et $\Delta 2$ (***id*** *"delta1"* et *"delta2"* dans le description en XML).
- ***<ght>***, ***<lht>*** ces sections définissent respectivement les *tâches hiérarchiques globales* (*global hierarchical task*) et les *tâches hiérarchiques locales* (*local hierarchical task*). La section ***<simulation>*** décrivant le couplage contiendra donc une section ***<ght>***. Les autres sections ***<simulation>*** contiendront des sections ***<lht>***, ou potentiellement une ***<mht>*** (*tâche hiérarchique principale* (*main hierarchical task*)) dans le cas de codes SPMD couplés entre eux. Dans le cas de codes SPMD, on utilise la description en *THP* des simulations au lieu de définir les *FTHL*.
- ***<remote>*** cette balise permet de définir les tâches distantes. Elles ne contiennent donc jamais de sous-tâches. L'attribut ***task_id*** doit correspondre à une THL (***<lht>***) ou à une sous-tâche d'une THP (***<mht>***). En effet, nous avons laissé la possibilité de faire pointer une tâche distante vers une sous-tâche d'une THP (***<mht>***). Cela permet dans le cas de couplage de type M-SPMD de pouvoir réutiliser les descriptions en THP des codes SPMD faites avant le couplage.
- ***<concurrent>*** cette section permet de définir les tâches concurrentes. Dans l'exemple, la tâche de type ***<concurrent>*** englobe deux sous-tâches distantes.

Au niveau de l'instrumentation des codes de simulation, il y a très peu de changement par rapport à la version précédente de EPSN. Les deux différences majeures sont la phase d'initialisation et les tâches distantes.

La phase d'initialisation est modifiée car il faut initialiser les *proxies* associés à chaque code et plus particulièrement le *master proxy*. Ce dernier peut être exécuté dans un processus séparé ou dans un thread dans le même processus qu'un des *proxies locaux*. De plus, il faut que chaque nœud de simulation et chaque *proxy local* spécifient le code auquel ils appartiennent via les fonctions *initMasterProxy*, *initProxy* ou *initNode*. Ces fonctions sont bloquantes ; pour éviter des interblocages lors de l'initialisation de un ou de plusieurs *proxies*, ainsi que d'un nœud dans le même processus, nous avons développé une fonction d'initialisation particulière *initProxyAndNode*. Nous présentons le prototype de cette fonction dans le listing 3.2. On notera qu'afin d'initialiser un *proxy* et un nœud EPSN, il faut spécifier le fichier XML contenant la description de la simulation et l'identifiant de la simulation couplée. Mais il faut également donner l'identifiant du code couplé correspondant au *proxy* et au nœud. Enfin, il faut spécifier le rang du nœud, le nombre total de nœuds, le numéro du code et le nombre de codes. Ces informations servent à l'initialisation des espaces de travail de ColCOWS. Le dernier paramètre de cette fonction permet de spécifier si un *master proxy* doit également être créé ou pas.

La seconde modification concerne les tâches distantes, elle consiste à ajouter des fonctions *beginLHT* et *endLHT* pour encadrer le code des fonctions dans les codes serveurs, et les fonctions *beginRemote* et *endRemote* pour encadrer les appels de ces fonctions dans les codes clients.

Nous avons volontairement limité les modifications apportées à l'API de EPSN afin d'offrir une compatibilité avec la version précédente. Ainsi, un code parallèle déjà instrumenté pourra toujours être piloté par EPSN2 sans avoir à être réinstrumenté, et pourra aussi être utilisé dans un couplage avec d'autres codes de simulation SPMD sans être modifié.

3.2.3 Implantation des différents algorithmes et schémas de communication

Le noyau d'EPSN contient les algorithmes de planification et la gestion des requêtes de pilotage et de monitoring. Nous avons déjà présenté au chapitre 2 les différents algorithmes

```
Status initProxyAndNode(//Nom de la simulation
                        const SimulationID & simulation_name,
                        //Fichier XML contenant la description
                        //de la simulation
                        const std::string & xml_filename,
                        //Rang du Noeud
                        unsigned int num_node,
                        //Nombre de noeud pour ce code
                        unsigned int nb_nodes,
                        //Nom du code
                        const CodeID & code_name = "",
                        //Rang du code
                        unsigned int num_code = 0,
                        //Nombre de codes
                        unsigned int nb_codes = 1,
                        //Initialise un master proxy si vrai
                        bool is_master = false);
```

Listing 3.2 – Prototype de la fonction *initProxyAndNode*

utilisés dans EPSN. Nous allons maintenant expliquer comment ils sont implantés dans EPSN2.

Pour implanter les algorithmes, nous nous sommes appuyés sur les algorithmes de planifications existants dans la version 1 de EPSN. Ces algorithmes permettent de planifier des traitements sur des simulations parallèles. Ce sont les mêmes algorithmes qui gèrent la planification des requêtes et les différents types de synchronisation des processus. Tous ces algorithmes varient uniquement au niveau du *proxy* ; par exemple lorsque le *proxy* demande une synchronisation, il peut demander à ce qu'elle soit bloquante ou non bloquante. Cela correspond soit à une pause, soit à une simple barrière « *à la MPI* ». Pour ce faire, la fonction *strongSynchronization* permet d'entamer le processus de synchronisation au niveau du *proxy*. Dans le cas d'une simulation parallèle, cette fonction fait appel à la fonction *weakSynchronization* puis se met en attente passive de la fin de la synchronisation. La fonction *weakSynchronization* envoie l'ordre de « *freeze* » afin de planifier une date de synchronisation, puis envoie cette date aux nœuds. Les nœuds informent le *proxy* lorsqu'ils ont atteint la date. Une fois la date de synchronisation atteinte par tous les nœuds, le *proxy* peut décider suivant le cas soit de maintenir les nœuds bloqués (pause), soit de libérer immédiatement les nœuds (barrière).

Nous allons à présent expliquer comment ces algorithmes ont été implantés pour des simulations distribuées. Dans les algorithmes présentés dans le chapitre précédent, nous avons utilisé des *coordinateurs*. Le rôle de ces *coordinateurs* sera rempli par les *proxies locaux* et le *master proxy*. Ainsi, les phases de planification débuteront par la réception d'une requête sur le *master proxy*. Nous allons commencer par décrire l'algorithme de planification simple, c'est à dire pour les requêtes de contrôle, puis nous verrons le cas plus complexe des requêtes de données.

3.2.3.1 Algorithme de planification des requêtes de contrôle

Cet algorithme, comme pour les simulations parallèles, correspond à un appel à la fonction *strongSynchronisation*. Dans ce cas, la fonction *strongSynchronisation* appelle également la fonction *weakSynchronisation* et se met en attente passive de la fin de la synchronisation sur les différents codes. La fonction *weakSynchronisation* gère la demande réelle de synchronisation des sous-codes. Pour cela, elle appelle la fonction *freezeSynchronisation* qui récupère les dates maximales atteintes par chaque code de simulation, via une réduction ColCOWS. En plus d'effectuer une réduction de la date sur chaque code, cet appel rend bloquants les points d'instrumentation des nœuds de la simulation. Une fois les dates maximales de chaque code

obtenues, la fonction ***weakSynchronisation*** déduit la date de planification et l'envoie aux différents nœuds via les *proxies locaux*. À la réception de cette date de synchronisation, les nœuds relâchent l'ordre de « *freeze* » et continuent leur exécution jusqu'à atteindre la date de planification. Cela met fin à la fonction ***weakSynchronisation*** au niveau du *master proxy* qui rend la main à la fonction ***strongSynchronisation*** pour que l'attente passive de la fin de synchronisation débute. Les codes vont prévenir le ***master proxy*** de façon asynchrone dès que leurs nœuds seront synchronisés.

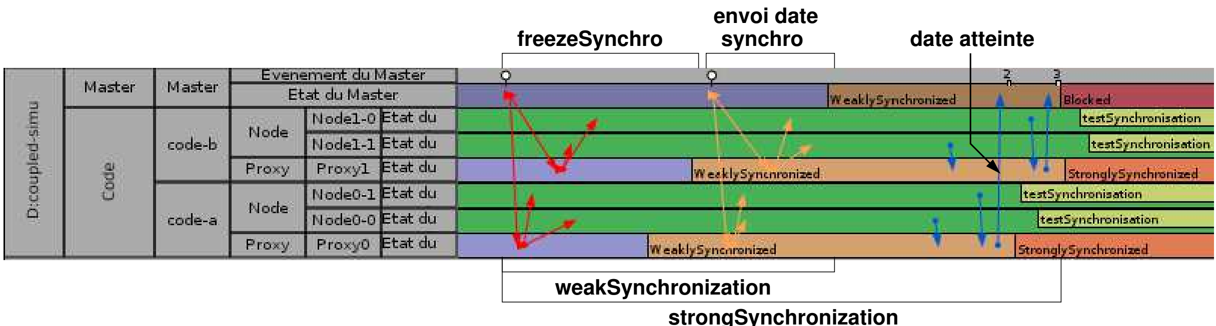


FIGURE 3.7 – Exemple de trace d'exécution lors d'une requête *pause*

La figure 3.7 illustre les communications et les changements d'états des nœuds et des *proxies* lors d'une requête « *pause* ». C'est une trace d'exécution obtenue sur une simulation de deux codes chacun possédant deux nœuds. Cette trace a été obtenue automatiquement lors de l'exécution de la simulation. La première communication collective (en rouge) correspond à l'ordre de « *freeze* ». En retour de cet appel, les *proxies locaux* débutent leur phase de synchronisation et passent dans un état « *WeaklySynchronized* ». Le *master proxy* finit de déterminer la date de synchronisation, l'envoie à tous les codes (en orange) et passe également dans l'état « *WeaklySynchronized* ». Pendant ce temps, les nœuds continuent leur exécution ; lorsqu'ils atteignent la date de synchronisation, ils en informent leurs *proxies* respectifs. Une fois tous les nœuds synchronisés, les *proxies* relaient l'information au *master proxy*. Une fois qu'un processus, nœud ou *proxy*, a atteint la date de synchronisation, il passe dans un état « *StronglySynchonized* » et il se met en attente passive jusqu'à la réception d'un ordre de libération de la synchronisation. La synchronisation s'achève quand le *master proxy* reçoit la confirmation du dernier *proxy local* comme quoi la synchronisation est atteinte. Il passe alors dans l'état « *Blocked* » et attend une requête *step* ou *play* pour relacher la synchronisation.

Cet enchaînement d'appels de fonctions et de changements d'état se produit lorsque la synchronisation se déroule normalement. Mais comme les nœuds continuent leur exécution tout au long de la planification, ils peuvent à tout moment atteindre un point d'instrumentation bloquant et donc stopper momentanément leur exécution. Si cela devait arriver, un algorithme local de « *rattrapage* » se déclenche. Les *proxies locaux* permettent aux nœuds les plus en retard de continuer leurs exécutions jusqu'à la date des nœuds les plus en avance, date qui correspond à la *date commune courante* propre au code. Pour ce faire, les nœuds notifient leur *proxy local* lorsqu'ils atteignent un point bloquant (flèche bleu sur la figure 3.8). Le *proxy* peut alors comparer cette date à la *date commune courante* et prévenir les nœuds les plus en retard qu'ils peuvent continuer leur exécution jusqu'au point ayant pour date la *date commune courante* (flèche marron sur la figure). Cela permet donc aux nœuds les plus en retard de « *rattraper* » les nœuds les plus en avance.

Sur la figure 3.8, on observe que les nœuds ***Node0-0*** et ***Node0-1*** du ***code-a*** envoient la

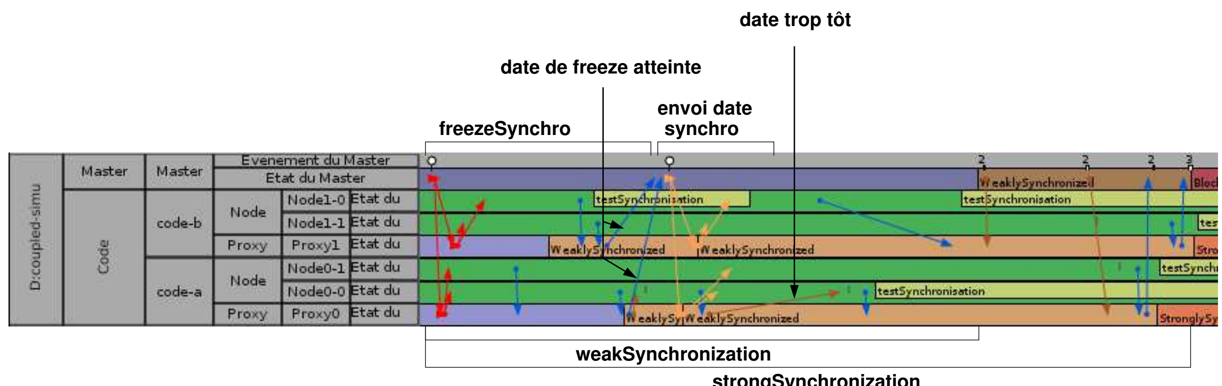


FIGURE 3.8 – Exemple de « *rattrapage* » lors d'une requête *pause*

date du point d'instrumentation qu'ils ont atteint. Puis le *proxy local* signale au nœud `Node0-0` qu'il s'est arrêté trop tôt. Le nœud va donc continuer son exécution et prévient à nouveau son *proxy* lorsqu'il atteint la nouvelle date d'arrêt. Entre temps, le *proxy local* a reçu la *date planifiée* du *master proxy*. Les traitements étant asynchrones, on peut noter que le *Proxy0* va prévenir le nœud qu'il est en retard. Car entre le moment ou le nœud prévient le *proxy* et ou le *proxy* traite cette notification la date de synchronisation a changé. Cette stratégie permet d'une part d'améliorer les temps de traitement des requêtes, et d'autre part de limiter les interblocages dûs à des erreurs d'instrumentations.

3.2.3.2 Algorithme de planification des requêtes de données

Le déroulement des requêtes de données est légèrement différent. Nous ne voulons pas synchroniser fortement les nœuds, nous devons juste planifier les traitements. Nous effectuons donc uniquement une étape de `weakSynchronisation`. Cette fonction, comme précédemment, réalise un appel à la fonction `freezeCurrentDate` qui détermine la *date planifiée* pour le traitement de la requête. Cette date est envoyée aux codes pour qu'ils puissent relâcher l'ordre de « *freeze* » et avancer au moins jusqu'à cette date. Ainsi, durant le temps que les *proxies* finissent de traiter la requête, les nœuds ont potentiellement le temps pour arriver à un point d'instrumentation bloquant. Puis les requêtes sont complétées : c'est-à-dire, le *master proxy* ajoute les informations relatives au code de simulation qui devra exécuter la requête et envoie la requête au *proxy* attaché au code concerné. Lorsque les *proxies locaux* reçoivent une requête, ils raffinent les conditions locales d'exécution avec les révisions courantes des données, c'est-à-dire les révisions maximales des données produites par les nœuds. Ces informations sont obtenues par les *proxies locaux* lors de l'ordre de « *freeze* ». Durant cet ordre, les nœuds ne renvoient pas uniquement leurs dates courantes, mais également les révisions courantes des données. Les informations sur les révisions étant propres à chaque code, elle sont conservées au niveau des *proxies locaux*. Les *proxies locaux* peuvent donc déterminer la révision maximale d'une donnée et l'ajouter à la requête demandant cette donnée avant de l'envoyer aux nœuds. Ensuite, les nœuds peuvent exécuter la requête dès que les conditions locales seront remplies. L'ordre de synchronisation est annulé une fois les requêtes envoyées. La figure 3.9 illustre cet algorithme. En violet foncé, on peut voir l'envoi de la requête de récupération d'une donnée du *code-a*, et en violet clair la fin de l'exécution de la requête notifiée par la réception d'un acquittement. Cet acquittement vient du *master proxy* car il est envoyé à ce dernier par le client lorsqu'il a fini de recevoir les données. Le *master proxy* transmet ensuite cet acquittement au proxy local et aux nœuds concernés.

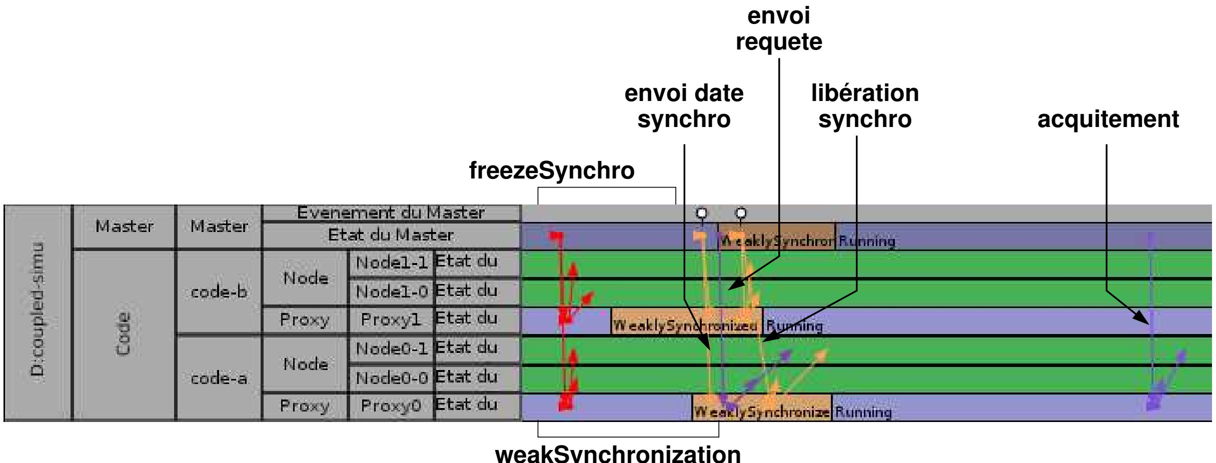


FIGURE 3.9 – Exemple de planification lors d'une requête *get* sur une donnée du *code-a*

Pour ce type de requête, il y a une difficulté supplémentaire lorsque la donnée est contenue dans une *THL* d'un serveur. Nous allons détailler ce cas particulier dans la section suivante.

3.2.3.3 Difficultés des simulations Client/Serveur

Au niveau de l'implantation du modèle pour les simulations Client/Serveur, il y a plusieurs difficultés : la vérification des conditions locales sur les *THL* et la concurrence des appels de fonctions sur les serveurs. La première est donc la vérification des conditions locales pour les requêtes de données. En effet, les conditions locales sont basées sur des dates complètes et des révisions de données. Comme les codes ne connaissent que les dates restreintes, on ne peut pas vérifier les conditions locales dans une *THL* (cas d'un code serveur) car il manque la date restreinte à la *THG*. Pour pouvoir vérifier ces conditions locales, les codes clients vont donc envoyer leur date courante au *master proxy* via les *proxies locaux* lors du passage sur le point d'instrumentation `beginRemote` ; ceci est donc fait au moment d'un appel à la méthode distante. Le *master proxy* relaie cette date au code appelé via son *proxy local*. Afin de ne pas trop interférer sur l'exécution de la simulation, le code serveur débute son exécution avant d'avoir reçu cette date. Ce n'est que lorsque les nœuds du code serveur auront besoin de vérifier des conditions locales qu'ils se mettront en attente de la date complète de la tâche appelante si elle n'a pas encore été reçue.

L'envoi de cette date introduit un nouveau problème ; en effet, dans le cas où les codes serveurs sont également des codes clients, ils devront également envoyer leurs dates. Les dates des codes serveurs étant des dates restreintes, le *master proxy* doit reconstituer les dates complètes à partir des dates restreintes qu'il a reçues avant de les fournir aux différents serveurs.

La figure 3.10 montre comment les dates complètes sont construites et envoyées aux codes concernés. La figure représente une simulation Client/Serveur dans laquelle le serveur `Serveur1` fait lui même appel à un serveur `Serveur2`. Ainsi, quand le code client appelle une *THL* du serveur `Serveur1`, il envoie en même temps sa date courante $d'_1 = 0.1.i.0$ restreinte à la *THG* au *master proxy*. Cette date étant une date complète, elle est envoyée sans modification au serveur sous le nom de d_1 dans notre exemple. Par contre, lorsque le serveur `Serveur1` appelle une *THL* du serveur `Serveur2`, il envoie également sa date courante $d'_2 = 0.1$ qui est une date restreinte. Le *master proxy* va devoir composer les dates d'_1 et d'_2 (*cf.* section 2.3.1.2) pour construire la

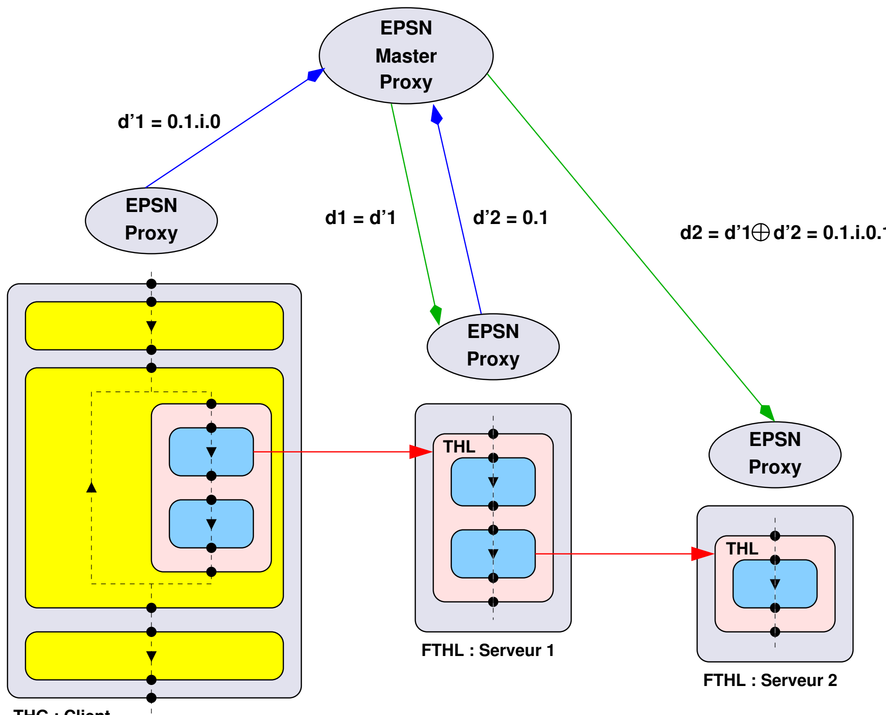


FIGURE 3.10 – Exemple d'envoi de dates globales dans le cas de Clients/Serveurs multiples

date globale $d_2 = d'_1 \oplus d'_2$ à envoyer au serveur ***Serveur2***. Pour cela, on concatène (au sens défini dans la section 2.3.1.2) la date $d'_1 = 0.1.i.0$ et la date $d'_2 = 0.1$, ce qui conduit à la date complète $d_2 = 0.1.i.0.1$

La seconde difficulté pour les simulations de type Client/Serveur concerne la concurrence d'appels de méthodes sur les serveurs. Nous ne pouvons pas faire des appels simultanés de THL sur un même serveur. Cela est dû au fait que l'on initialise une seule fois l'environnement sur chaque serveur : le *proxy local* associé à un serveur ainsi que les nœuds EPSN sont initialisés au lancement du serveur. Par conséquent, les contextes EPSN et RedGRID sont communs à toutes les *THL* d'un serveur. Les appels de plusieurs méthodes simultanément sur un même serveur entreraient donc en conflit au niveau de la plate-forme EPSN et de la bibliothèque RedGRID. Pour remédier à cela, nous pourrions réinitialiser les contextes EPSN et RedGRID à chaque appel sur une *THL*, mais cela s'avèrerait bien trop coûteux. Par conséquent, nous avons préféré opter pour une solution consistant à empêcher que plusieurs *THL* d'un même serveur s'exécutent en même temps. Pour cela, les appels à la fonction ***beginRemote*** sont bloquants dans le cas où l'appel à distance (encadré par les fonctions ***beginRemote*** et ***endRemote***) devrait être fait sur une *THL* d'un serveur qui est déjà en train d'exécuter une *THL*. En conséquence, EPSN offre la possibilité d'attendre la fin de l'exécution d'une *THL* dans le code appelant en rendant la fonction ***endRemote*** synchronisante.

3.2.3.4 Extensions des schémas de communication

Nous allons maintenant expliquer comment nous avons étendu les schémas de communication pour les simulations distribuées. Pour commencer, nous détaillerons la phase de connexion d'un client à une simulation couplée, en rappelant tout d'abord le fonctionnement dans le cas des simulations SPMD.

Dans le cas des simulations SPMD avec EPSN1, le proxy inscrit lors du déploiement la simulation dans un service de nommage CORBA pendant la phase d'activation du *workspace* ColCOWS de la simulation. En consultant le service de nommage, il est ainsi possible de lister toutes les simulations pilotables en cours d'exécution. Les clients peuvent alors facilement trouver les différentes simulations et obtenir les informations nécessaires à la connexion. Dans le cas des simulations distribuées, les *proxies locaux* vont inscrire le code dont ils sont responsables et le *master proxy* inscrira la simulation distribuée. Les clients ont donc le choix, soit de se connecter à la simulation distribuée, soit de se connecter directement à un code particulier composant la simulation. Lorsqu'un client se connecte directement à un code, la connexion se déroule alors exactement comme pour un code SPMD. Par contre, s'il se connecte à la simulation distribuée, la phase de connexion est plus complexe. En effet, le client va tout d'abord se connecter au *master proxy* qui, en réponse, va envoyer la liste des codes auxquels le client doit également se connecter (les codes composant la simulation). Le client se connecte alors aux différents codes et à chaque connexion, on associe un numéro de connexion unique, appelé *ConnectionID*. Ensuite, le client envoie au *master proxy* les différents *ConnectionID* qu'il a obtenu lors de la phase précédente. Ces informations serviront par la suite pour le traitement des requêtes.

Toutes les informations nécessaires au traitement des requêtes sont donc définies après la phase de connexion. Le client a connaissance de l'ensemble des codes et de leurs descriptions, ces dernières étant échangées lors de la connexion des *workspaces ColCOWS* . Décrivons à présent les schémas de communications lors du traitement des requêtes de pilotage, tout d'abord pour les requêtes de contrôle puis pour les requêtes de données.

Pour les requêtes de contrôle, les schémas de communication sont assez simples. Lors de la réception d'une requête *stop* sur le *master proxy*, ce dernier va entamer une phase de synchronisation. Ainsi, le schéma de communication n'est autre que celui de la synchronisation détaillée précédemment.

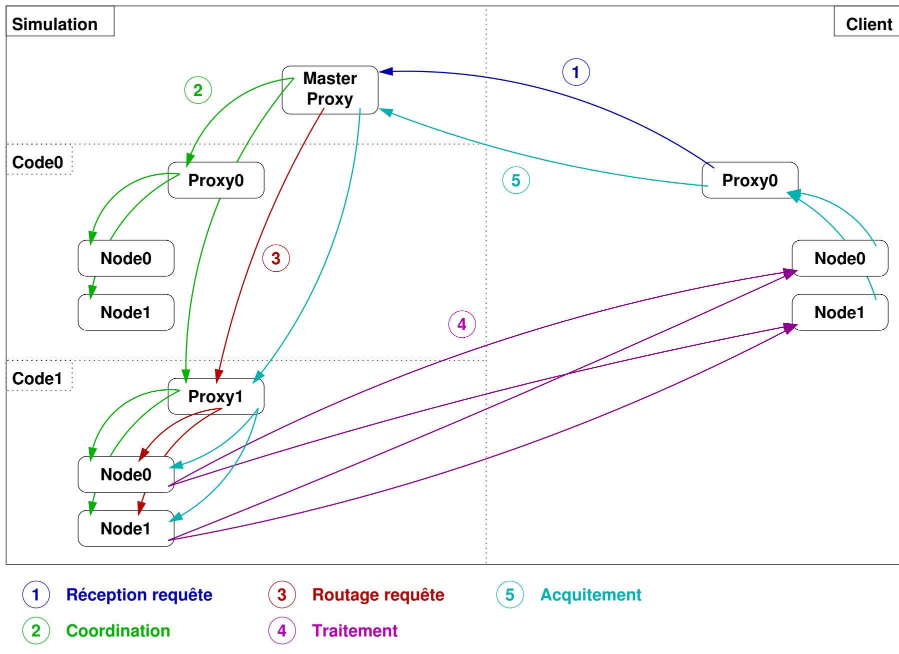


FIGURE 3.11 – Schéma de communication lors d'une requête *get*

Pour les requêtes de données, cela s'avère plus complexe. La figure 3.11 illustre le schéma

de communication d'une requête de données de type *get*. Cette requête demande une donnée localisée sur un code particulier. Comme pour les requêtes de contrôle, le schéma débute par l'envoi par le client de la requête au *master proxy* (1). Le client peut demander une liste de données, et à chaque donnée l'API client d'EPSN associe le *ConnectionID* qui correspond au code qui détient la donnée ; cela permet au *master proxy* de « *router* » correctement la requête vers le bon code. Par la suite, les nœuds utiliseront ce *ConnectionID* pour envoyer les données via le bon canal. À la réception de la requête, le *master proxy* débute une phase de coordination (2). Comme nous l'avons vu, cette phase implique tous les processus de tous les codes de simulation, même si la requête ne concerne qu'un seul code. Puis une fois la date de traitement restreinte à la *THG* déterminée, le *master proxy* envoie la requête aux codes détenants les données demandées (3). Une fois que les *proxies locaux* concernés ont reçu la requête, ils la relaient aux nœuds. Les nœuds pourront lors du passage sur les points d'instrumentation vérifier les conditions locales afin de vérifier si la requête est exécutable ou pas. Lorsque les conditions locales sont vérifiées, la requête est prête à être exécutée (état *ready*). La requête est donc exécutée en « *tâche de fond* » par des *threads* internes à la plateforme EPSN que nous appelons *workers*. Les *workers* ont pour rôle de vérifier s'il y a des requêtes *ready* et de les exécuter. Le nombre de workers est limité sur chaque nœud ; les requêtes sont donc placées dans une file d'attente et exécutées tour à tour dès qu'un *worker* se libère. Le nombre de *workers* est configurable avant le lancement de la simulation. Dans le cas d'une requête de données, ce sont donc les workers qui se chargent d'effectuer les envois de données asynchrones (4) en utilisant la bibliothèque RedCORBA. Une fois que le client a reçu les données, il envoie un acquittement à la simulation (5). Le *master proxy* envoie l'acquittement aux codes qui mettent fin au traitement de la requête et libèrent la simulation dans le cas où elle s'était bloquée en attente de la fin du traitement. Ce blocage est utile dans le cas où les conditions locales ne seraient plus vérifiées. Le client peut ainsi faire attendre la simulation entre le traitement de plusieurs requêtes en retardant l'acquittement. Cela permet par exemple dans le cas de requêtes permanentes, d'assurer que le client ait fini ses post-traitements de visualisation avant de recevoir la *révision* suivante de la donnée.

Nous venons de présenter les principales extensions faites dans la gestion des simulations de la plate-forme EPSN. Nous allons à présent voir les modifications que cela implique au niveau des clients de pilotage. Pour cela, nous allons présenter les différents clients fournis avec la plate-forme EPSN.

3.3 Client de pilotage et de visualisation

Dans la section précédente, nous avons expliqué comment intégrer des simulations distribuées dans la plate-forme EPSN ; à présent, nous allons décrire la partie clients de pilotage et de visualisation de EPSN.

3.3.1 Clients EPSN

EPSN2 permet de piloter des simulations distribuées en instrumentant et en décrivant les codes. Mais pour piloter pleinement les simulations, il faut avoir un moyen d'envoyer des ordres de pilotage. Pour cela, le noyau d'EPSN possède une API permettant de construire des clients de pilotage et de visualisation. Cependant, nous proposons aux utilisateurs un client de pilotage générique, qui permet de piloter toute simulation préalablement instrumentée. Ce client, nommé *Simone* (Simulation Monitor for EPSN), est construit à partir de plugins QT. Ces plugins ont pour but de pouvoir être réutilisés pour construire d'autres clients plus spécifiques. Ils ont permis

par exemple de construire des plugins dédiés à Paraview afin de pouvoir visualiser « *à la volée* » les données des simulations directement dans Paraview.

L'API client d'EPSN permet de plus de construire des clients parallèles afin de pouvoir mieux traiter les gros volumes de données pouvant être générés par les simulations. Pour cela, l'architecture des clients est similaire à celle des simulations : il y a un *proxy* et des nœuds regroupés dans un *workspace* ColCOWS. Les nœuds servent essentiellement à gérer les données en parallèle et le *proxy* sert à « *orchestrer le tout* ».

Nous allons présenter les différentes briques de base permettant de construire des clients de pilotage et de visualisation, puis nous présenterons les clients séquentiels et parallèles de EPSN.

3.3.1.1 Briques de base des clients EPSN

Les briques de base sont de deux types : nous avons tout d'abord les plugins QT [119], permettant de construire des interfaces utilisateurs basées sur la bibliothèque QT pour piloter des simulations et nous avons les sources VTK [141] permettant de visualiser les données à l'aide de pipelines de visualisation VTK.

Plugins QT – Les plugins QT permettent de mettre à la disposition des développeurs un moyen simple de construire des clients EPSN en QT. Ces plugins fournissent différentes informations : une représentation graphique des tâches hiérarchiques, la liste des données instrumentées et la liste des interactions disponibles. Ils permettent également d'interagir avec les simulations en émettant des requêtes et en permettant de surveiller l'état de ces requêtes. Pour finir, dans le cas de clients séquentiels, ils permettent également de visualiser les données brutes de la simulation et de les modifier.

Sources VTK – Elles permettent de convertir la description des données RedSYM en une description VTK.

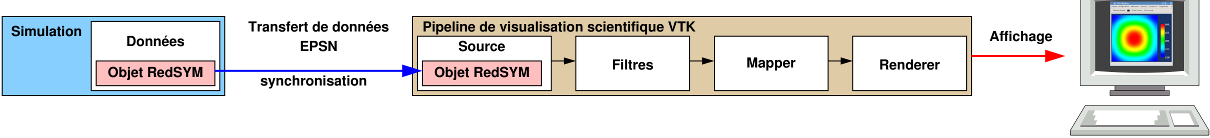


FIGURE 3.12 – Pipeline de visualisation scientifique à base de sources de données VTK/RedSYM

Les sources VTK/RedSYM peuvent ensuite être utilisées dans un pipeline de visualisation graphique construit avec la bibliothèque VTK (voir figure 3.12). Ainsi, les données des simulations pilotées peuvent être traitées par la bibliothèque VTK et cela avec un coup de développement quasiment nul. VTK utilise ses propres types de données et nous devons donc convertir les données récupérées dans le client dans les types natifs de VTK. Pour certaines sources (maillages et points), cela se limite à construire une description pour que VTK puisse lire les données directement dans la mémoire du client. Pour d'autres, cela est plus complexe car les données en mémoire ne sont pas compatibles avec les formats de données VTK. Dans ce cas, les sources VTK/RedSYM se chargent de convertir les données au bon format. Le tableau 3.1 récapitule les différentes données prises en charge par la bibliothèque RedSYM et les types de données VTK utilisés pour les représenter.

À partir de l'API EPSN, des plugins QT et des sources VTK, il est alors possible de construire

Type de données	Type RedSYM	Type VTK	Représentation
Points ou atomes	Points	Polydata	
Maillage 2D (triangle ou quadrangle)	Mesh	Polydata	
Grille creuse	Grid		
Maillage 3D (tetrahèdre ou hexahèdre)	Mesh	UnstructuredGrid	
Grille structurée dense sans coordonnées	Grid	ImageData	
Grille structurée dense avec coordonnées	Grid	StructuredGrid	

TABLE 3.1 – Tableau de correspondance entre les types de données RedSYM et VTK

des clients de pilotage et de visualisation relativement complexes. Par la suite, nous distinguerons deux types de clients : les clients séquentiels et les clients parallèles.

3.3.1.2 Clients séquentiels

Les clients séquentiels ont la particularité d'avoir un *proxy* et un seul nœud EPSN généralement dans le même processus. Les données sont donc centralisées sur un seul nœud de traitement dans le client. Cela permet de construire plus facilement un client car les données du nœud et du *proxy* sont toutes disponibles dans le même processus.

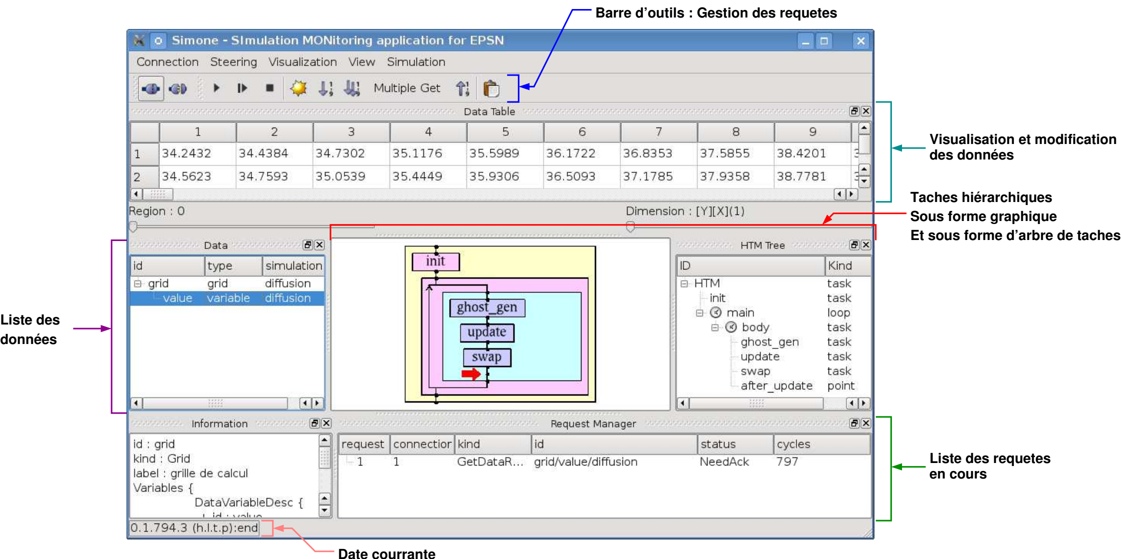


FIGURE 3.13 – Vue d'ensemble du client Simone développé à l'aide des plugins QT

La plate-forme EPSN fournit un client séquentiel générique. Ce client, illustré par la figure 3.13, se nomme Simone (Simulation Monitor for Epsn). Il est construit à partir des plugins QT ainsi que des sources VTK. Il permet donc d'interagir pleinement avec les simulations pilotables ainsi que de visualiser de façon sommaire les données. Le client Simone permet de plus de se connecter simultanément à plusieurs simulations. D'autres clients séquentiels plus spécifiques ont également été développés, comme le client réalisé pour visualiser les résultats de la simulation couplée basée sur les codes DLPoly et Siesta (*cf.* section 1.1.2.2). Ce client se nomme MonIQA (Monitoring Interface for QM/MM Applications) et il utilise la bibliothèque Tulip [134] pour visualiser les graphes. La figure 3.14 présente une vue d'ensemble de MonIQA.

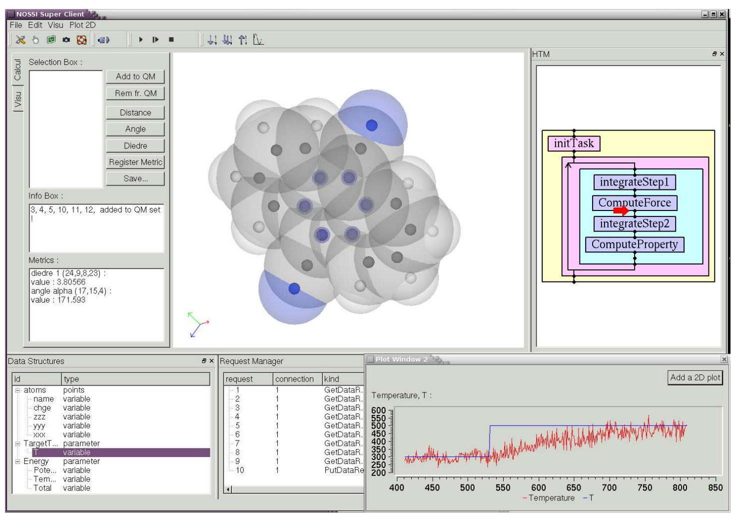


FIGURE 3.14 – Vue d'ensemble du client MonIQA

Le client Simone utilise, pour visualiser simplement les données, le plugin QT permettant de voir les données sous forme de « *feuille de données* » à la manière d'un tableur. Il utilise également les sources VTK fournies par RedGRID, ainsi qu'un pipeline de visualisation sommaire qui permet de visualiser les données sans aucun post-traitement. Cependant, la mise à jour du pipeline VTK peut poser des problèmes.

En effet, VTK utilise un pipeline de type *demand-driven* (*cf.* section 1.2). Par conséquent, le pipeline de visualisation scientifique n'est mis à jour que sur un évènement venant du client, comme un clic de souris ou un appui sur une touche du clavier. Comme les nouvelles versions des données sont envoyées par la simulation de façon asynchrone, il faut donc un moyen de mettre à jour le pipeline de visualisation lorsque ces données sont réceptionnées. Pour ce faire, on utilise des évènements particuliers de type *timer* : ces évènements nous permettent, dans la boucle d'interaction principale du client, de vérifier régulièrement si une nouvelle version a été réceptionnée ou pas. Dans un tel cas, on force la mise à jour du pipeline, ce qui a pour effet de produire une nouvelle image caractérisant l'état courant de la simulation.

Pour les requêtes permanentes, cette méthode introduit un autre problème. En effet, si les données arrivent de façon périodique, on ne peut pas assurer qu'une mise à jour du pipeline ne se fasse pas en même temps que la réception d'une donnée comme nous l'avons expliqué dans la section 2.3.2.3. Pour remédier à ce problème, nous avons proposé la solution de l'acquittement au plus tard qui se traduit par un *explicit ack*. Par défaut, lorsqu'un client émet une requête, elle est acquittée au plus tôt de façon automatique par la plate-forme EPSN à la réception des données ; si ce client veut pouvoir l'acquitter au plus tard, il doit explicitement le spécifier au

moment de la création de la requête, d'où l'*explicit ack*.

Nous avons vu comment nous pouvons assurer la cohérence des données dans un client séquentiel grâce à la méthode de l'*explicit ack* (ou acquittement au plus tard) qui rend donc la main à la simulation après les traitements. Nous allons à présent voir comment assurer la cohérence des données pour un client parallèle.

3.3.1.3 Clients parallèles

Les clients parallèles, par opposition aux clients séquentiels, ont plusieurs nœuds EPSN. Les données réceptionnées dans le client sont donc réparties sur les différents nœuds du client. EPSN fournit des plugins Paraview permettant de s'intégrer dans cet outil de visualisation scientifique. Paraview nous permet de faire de la visualisation scientifique en parallèle en répliquant le pipeline de visualisation ; ainsi le développement de clients parallèles de visualisation est fortement facilité. Nous pouvons également construire des clients parallèles directement à l'aide des sources VTK qui sont prévues pour fonctionner en parallèle. Dans les deux cas, cela revient à associer une source VTK à chaque nœud du client.

Comme nous l'avons vu dans la section 2.3.2.3, les clients parallèles introduisent un nouveau problème de cohérence dû à la répartition des données sur les différents nœuds. Nous avons dans ce cas deux solutions : la méthode de l'acquittement au plus tard ou l'algorithme de *lock global*.

Tout d'abord, nous allons expliquer la mise en œuvre de la méthode de l'*explicit ack*. Cette méthode est plus simple car elle n'intervient que sur le *proxy*. En effet, comme le montre la figure 3.15, une fois que le *proxy* est notifié par les nœuds clients de l'arrivée d'une nouvelle version de données (2), il peut forcer la mise à jour des pipelines de visualisation scientifique (3). Une fois les traitements des données effectués (4), le rendu (5) et l'affichage (6) réalisés, le *proxy* acquitte la requête (7).

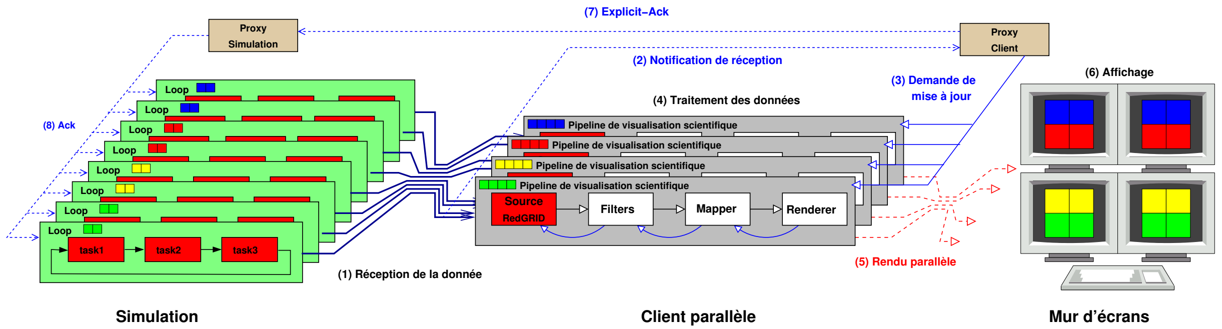


FIGURE 3.15 – Méthode de l'*explicit ack* dans un client de visualisation parallèle

La méthode du *lock global* suit l'algorithme du même nom décrit dans le chapitre précédent (*cf.* section 2.3.2.3). Dans ce cas, le rôle du coordinateur est joué par le proxy EPSN du client. Afin de verrouiller les données, nous utilisons sur les nœuds EPSN les verrous sur les données disponibles dans la bibliothèque RedGRID. En effet, RedGRID permet de limiter l'accès aux objets RedSYM en lecture et/ou en écriture.

Nous avons donc à notre disposition deux méthodes pour assurer la cohérence des données dans le client : l'*explicit ack* qui bloque la simulation jusqu'à ce que le traitement parallèle de la donnée pour la visualisation soit fini, et le *lock global* qui empêche la réception d'une nouvelle donnée tant que le traitement parallèle de la précédente n'est pas fini.

3.4 Conclusion

La plate-forme EPSN2 permet de piloter des simulations distribuées de façon générique. Cette plate-forme propose une architecture flexible et dynamique basée sur des technologies reconnues et très largement répandues telles que XML pour la description des simulations, CORBA pour les communications internes à la plate-forme, QT pour la conception de clients et VTK & Paraview pour la visualisation séquentielle ou parallèle des données. La plate-forme implante pleinement les modèles de description des simulations et des données présentés dans le chapitre précédent et les algorithmes de coordination des requêtes de contrôle. Par contre, pour les requêtes de données, elle n'implante que le cas des simulations M-SPMD, le cas des simulations Client/Serveur n'étant pas totalement intégré. En effet, dans le cas Client/Serveur, il faut prendre en compte le cas des dates complètes qui ne sont pas présentes sur les codes serveurs. Ces difficultés des algorithmes Client/Serveur ont donc été bien identifiées mais à ce stade du développement de la plate-forme, les solutions ne sont que partiellement implantées.

Nous allons à présent évaluer les développements effectués dans la plate-forme EPSN2. Pour ce faire, il nous faudra évaluer toutes les étapes de la plate-forme allant du déploiement aux traitements des requêtes. Nous verrons également comment se comporte la plate-forme pour une « *vraie* » simulation distribuée.

Chapitre 4

Évaluation de la plate-forme de pilotage EPSN2

Sommaire

4.1 Introduction **94**

4.2 Évaluation de la bibliothèque ColCOWS **96**

4.2.1 Activation d'un workspace 96

4.2.2 Communications collectives 97

4.3 Test de validation pour une simulation M-SPMD **99**

4.3.1 Initialisation de la plate-forme 99

4.3.2 Algorithmes de coordination 100

4.3.3 Client de visualisation parallèle 103

4.4 Test de validation pour des simulations distribuées réelles **104**

4.4.1 LibMultiScale : simulation couplée de type M-SPMD 105

4.4.1.1 Cas-test 2D de propagation d'onde dans un crystal d'Argon . . 105

4.4.1.2 Cas test 3D de contact glissant de surfaces rugueuses 108

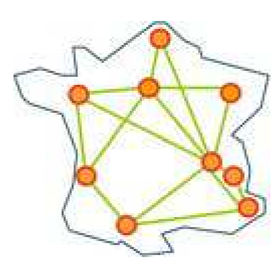
4.4.2 Couplage DLPoly/Siesta 109

4.5 Conclusion **111**

4.1 Introduction

Avant de présenter les performances de la plate-forme EPSN2, nous allons décrire les configurations matérielles ayant servi à réaliser nos expériences. Nous avons utilisé pour ce faire la grille de calcul *Grid'5000* [56] ainsi que notre cluster local de visualisation. Nous utiliserons différents clusters de cette grille afin de faire les tests de passage à l'échelle de l'environnement EPSN, les clients de pilotage étant exécutés sur le cluster graphique local.

La grille de calcul *Grid'5000* est une grille regroupant des clusters présents sur 9 sites géographiques en France (voir figure 4.1). L'ensemble des 4 sites regroupe plus de 5000 cœurs. Ces sites sont interconnectés via le réseau RENATER-5 (RÉseaux NAtional de télécommunications pour la Technologie, l'Enseignement et la Recherche) [123]. Le réseau RENATER-5 est un réseau à base de fibres optiques reliant les différentes universités françaises. Le projet *Grid'5000* bénéficie d'un réseau dédié avec une capacité de 10Gbit/s sur le réseau RENATER.



Site	Processeurs	Cœurs
Bordeaux	424	650
Grenoble	68	272
Lille	290	618
Lyon	260	268
Nancy	518	1310
Orsay	684	684
Rennes	326	908
Sophia	356	568
Toulouse	276	436
Total	3202	5714

FIGURE 4.1 – Description de la plate-forme Grid'5000

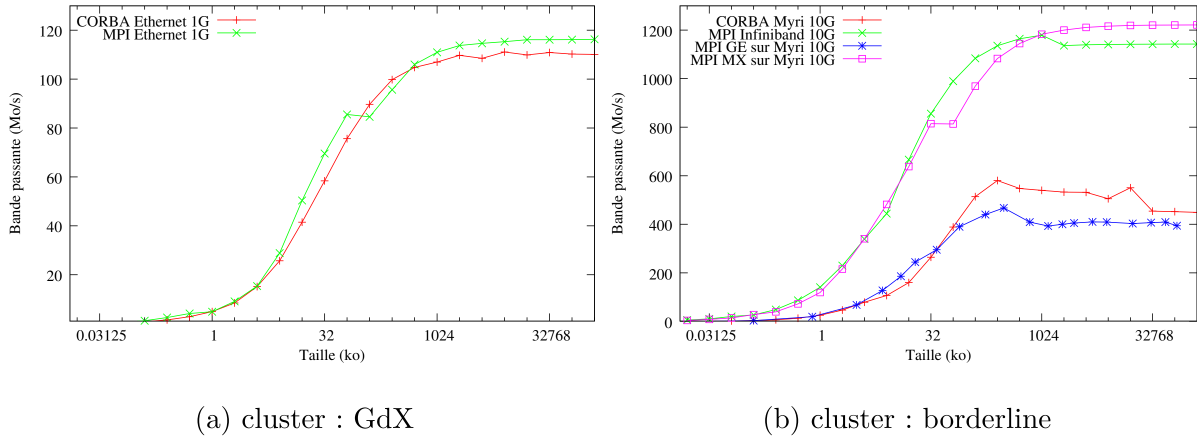
Pour nos expériences, nous n'utiliserons pas tous les sites de *Grid'5000* ; nous utiliserons essentiellement le site d'Orsay pour les tests de passage à l'échelle et celui de Bordeaux pour les expériences de pilotage de simulations. Le site d'Orsay dispose d'un cluster de 312 nœuds bi-Opteron (AMD, 64bits) dont 186 sont cadencés à 2.0 *Ghz* et les 126 autres à 2.4 *Ghz*. Ces nœuds possèdent 2 *Go* de mémoire partagée et sont interconnectés via un réseau Gigabit Ethernet et un réseau Myrinet 10G. Ce cluster se nomme *GdX*.

Le site de Bordeaux dispose de plusieurs clusters ; nous en utiliserons deux en particulier. Le premier est un cluster de 93 bi-Opteron dual-core cadencés à 2.6 *Ghz* nommé *Bordereau*. Les nœuds de ce cluster possèdent 4 *Go* de mémoire partagée et sont interconnectés par deux réseaux Giga Ethernet. Le second, nommé *borderline*, est un cluster de 10 quadri-Opteron dual-core cadencés à 2.6 *GHz* avec 32 *Go* de mémoire partagée. Ce cluster possède 3 réseaux, un réseau Giga Ethernet, un réseau Myrinet 10G et un réseau Infiniband 10G.

Notre cluster graphique, nommé *burdigala* est composé quant à lui de 6 PCs bi-Opteron dual-core cadencé à $2,6$ *GHz*. Ces PC ont 8Go de mémoire et sont interconnectés par deux réseaux, un Giga Ethernet et un Infiniband. De plus, ils possèdent des cartes graphiques nVidia Quadro FX 4500 X2. Quatre des machines de ce cluster sont équipées d'un écran, ces écrans formant un mur d'écran 2×2 (voir photo 4.2). Ce cluster est connecté à l'aide d'une fibre optique aux clusters du site de Bordeaux. Nous disposons donc d'une connexion 1G/10G entre notre cluster *burdigala* et le cluster *borderline*.

FIGURE 4.2 – Mur d'images du cluster graphique

Pour finir cette introduction, nous allons présenter quelques courbes de référence utiles à l'analyse des résultats qui suivront. Les figure 4.3 (a) et (b) présentent les débits réseau obtenus sur les différents clusters que nous allons utiliser. Ces mesures sont obtenues en effectuant des transferts point à point avec la bibliothèque MPICH2 et avec le middleware OmniORB4. Ce sont les implantations respectives des spécifications MPI et CORBA que nous allons utiliser.



(a) cluster : GdX

(b) cluster : borderline

FIGURE 4.3 – Courbes de référence : CORBA, MPI

Comme le montre la figure 4.3 (a), les bandes passantes obtenues pour CORBA et MPI sont assez proches sur les réseaux Giga Ethernet du cluster *GdX* d'Orsay. Le débit maximal obtenu sur ce réseau avec OmniORB4 est de 110 Mo/s (latence 153 μs) contre 116 Mo/s (latence 96 μs) pour MPI. Cette implantation de CORBA est donc quasiment aussi performante que MPI sur ce type de réseau. Sur le cluster *borderline* de Bordeaux, figure 4.3 (b), nous avons fait les mesures de OmniORB4 sur le réseau Myrinet. Ce réseau a une sur-couche permettant

de faire passer du trafic TCP/IP via un *driver* Ethernet et nous pouvons donc l'utiliser avec CORBA qui se limite à ce type de communication. Nous notons que le débit maximal que nous obtenons est de 580 Mo/s (latence 40 μs) ; ce résultat est loin de la bande passante théorique de ce type de réseau (1280 Mo/s). Ceci est dû au protocole *TCP/IP* qui n'est pas adapté à ce type de réseau ; cela vient *a priori* de la taille des fenêtres TCP. Par ailleurs, l'implantation MPI utilisant des *sockets* TCP/IP n'est guère plus performante, avec un débit maximal de 467 Mo/s (latence 4 μs). Si on utilise le *driver* MX avec MPI, qui est le driver natif pour les cartes Myrinet, nous avons des valeurs bien plus correctes pour ce type de réseau, soit un débit maximum de 1221 Mo/s et une latence de 4 μs. Malgré la qualité des résultats des communications CORBA sur les cartes réseau Myrinet, nous utiliserons tout de même ce réseau car les messages internes à la plate-forme EPSN sont de petits messages ; nous aurons donc essentiellement le gain dû à la faible latence de ce réseau (40 μs contre 153 μs sur un réseau Giga-Ethernet).

Dans la suite de ce chapitre, nous allons présenter les différentes expériences et résultats obtenus avec la plate-forme EPSN2. Pour ce faire, nous allons tout d'abord évaluer la bibliothèque ColCOWS qui est utilisée pour effectuer toutes les communications collectives. Puis nous allons mesurer les performances de EPSN2 à l'aide d'une simulation « *test* ». Enfin, nous verrons comment se comporte la plate-forme sur une simulation réelle. Toutes ces mesures se feront sur des simulations de type M-SPMD car les simulations de type Client/Serveur ne sont pas totalement prisent en compte par la plate-forme dans sa version actuelle.

4.2 Évaluation de la bibliothèque ColCOWS

Nous allons commencer par évaluer la bibliothèque ColCOWS, qui comme nous l'avons vu précédemment est utilisée dans la plate-forme EPSN2 pour les communications collectives. Nous allons donc mesurer le coût des communications de type *gather* et *reduce*. Pour ce faire, nous allons activer un espace de travail ColCOWS composé de N nœuds et comparer les temps mis pour récupérer une variable depuis chaque nœud sur le nœud 0 de façon hiérarchique, et ce par un appel à la méthode *gather* sur chaque nœud. Puis, nous allons faire de même dans le cas de la réduction (*reduce*) d'une variable. Pour toutes ces mesures, nous allons utiliser un nœud physique, soit une carte réseau par nœud ColCOWS. Cette limitation nous permet d'avoir une homogénéité du réseau utilisé pour les communications entre les nœuds ColCOWS.

4.2.1 Activation d'un workspace

Afin de pouvoir effectuer des communications collectives entre des nœuds ColCOWS, nous devons avant tout activer un workspace. Cette étape consiste à créer le *workspace* ColCOWS en échangeant les références des objets CORBA détenues par les nœuds ColCOWS. On doit noter également que l'étape d'activation est effectuée lors de l'initialisation de la plate-forme EPSN. Nous allons donc mesurer le temps mis pour activer un espace de travail (voir figure 4.4) en fonction du nombre de noeuds. Ces temps nous permettront par la suite d'analyser plus finement les temps d'initialisation de la plate-forme EPSN.

L'activation d'un espace de travail est la seule étape collective de la bibliothèque ColCOWS qui n'utilise pas de communications hiérarchiques, car c'est durant cette étape que les nœuds prennent connaissance les uns des autres. Par conséquent, comme le montre la figure 4.4, le temps d'activation est linéaire par rapport au nombre de nœuds. Les multiples pics sont dûs à l'utilisation du service de nommage. Le service de nommage est un service CORBA permettant

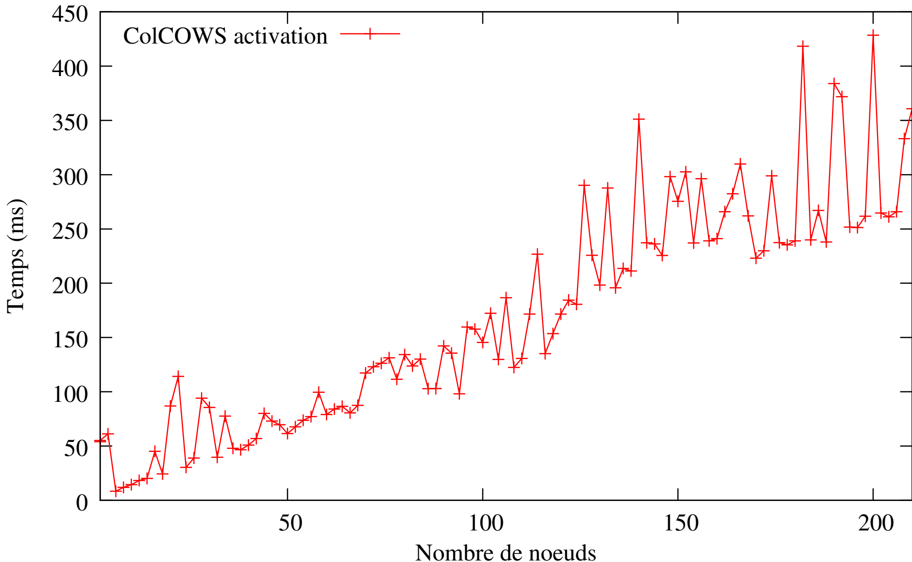


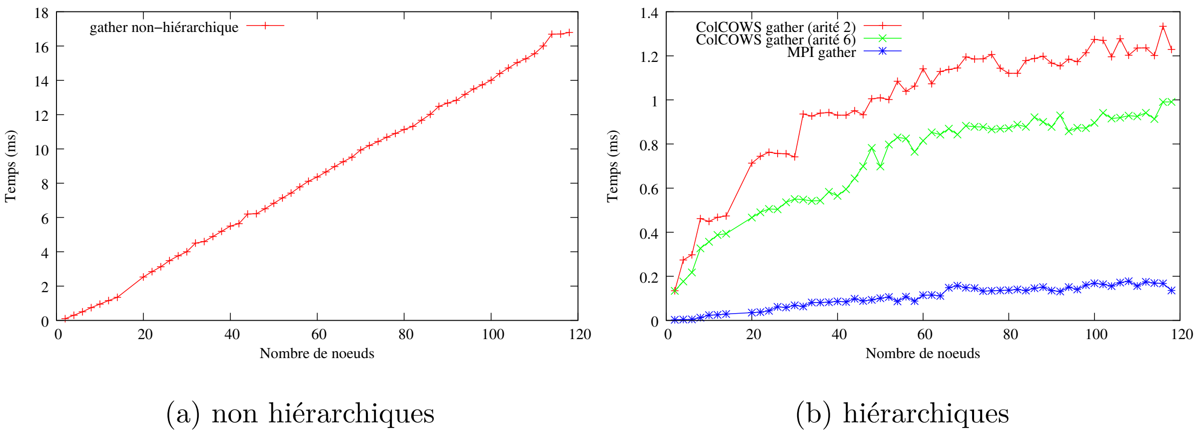
FIGURE 4.4 – Temps d'activation d'un workspace ColCOWS

deux choses : enregistrer des références d'objets ou résoudre des références. Il est donc fortement sollicité durant la phase d'activation de ColCOWS :le nœud 0 s'enregistre avec l'identifiant *Workspace_ID*, puis tous les autres nœuds essaient de résoudre cet identifiant. Si trop de nœuds tentent la résolution en même temps, il en résulte une exception CORBA. Les nœuds qui reçoivent cette exception attendent un certain temps paramétrable avant de retenter la résolution du *Workspace_ID*. Le nombre maximum d'essais de résolution que les nœuds effectuerons est fonction du nombre de nœuds dans l'espace de travail car le nombre de collisions augmentent avec le nombre de nœuds.

4.2.2 Communications collectives

Une fois l'espace de travail activé, nous pouvons effectuer des communications collectives de type *gather* ou *reduce*. Nous avons donc effectué ces deux types de communication sur des *workspaces* de différentes tailles et avec différentes arités pour l'arbre définissant les communications à faire. La donnée que nous transférerons durant ces tests est, une variable de type entier soit 4 octets. Pour ces mesures, nous nous sommes limités aux nœuds homogènes du cluster GdX cadencés à 2.4 GHz. Nous avons fait les mesures pour un plus grand nombre de nœuds, mais l'implantation de MPI que nous utilisons change l'algorithme de *gather* utilisé et n'est plus logarithmique comme il devrait l'être. Ce changement d'algorithme vient *a priori* du fait que l'implantation de MPI détecte une certaine hétérogénéité.

Les graphiques 4.5 (a) et (b) présentent les résultats pour des communications de type *gather*. Le graphique 4.5 (a) correspond aux communications non hiérarchiques utilisées dans la plateforme EPSN1, et le graphique 4.5 (b) présente les communications faites de façon hiérarchique qui sont utilisées dans la version 2 de EPSN. Quel que soit l'arité de l'arbre de communication utilisé pour les communications hiérarchiques, les résultats sont toujours bien meilleurs que pour la version non hiérarchique. Le temps maximal pour un *gather* hiérarchique sur 120 nœuds est de 1.23 ms (1.70 ms sur 210 nœuds) avec une arité de 2 et de 0.99 ms avec une arité de 6 (1.24 ms sur 210 nœuds), contre 16.8 ms (32.9 ms sur 210 nœuds) pour la version non hiérarchique. Les résultats sur un grand nombre de nœuds sont donc environ 20 fois meilleurs. Ces résultats



(a) non hiérarchiques (b) hiérarchiques

FIGURE 4.5 – Communications collectives : *gather*

étaient prévisibles car on passe d'un algorithme ayant une complexité linéaire en nombre de nœuds à un algorithme de complexité logarithmique. On peut également noter que c'est avec une arité de 6 que l'on obtient de meilleurs résultats. En fait, c'est un bon compromis entre le nombre de communications séquentialisées sur les nœuds et la hauteur de l'arbre. Nous avons testé d'autres arités ; afin de ne pas surcharger les graphiques, nous n'avons représenté que le meilleur résultat ainsi que l'arité de 2 que nous prenons comme référence. L'arité 2 nous sert de référence car l'implantation de MPI que nous utilisons, s'appuie sur un arbre bi-nomial pour effectuer les communications collectives.

Il est à noter que nos résultats, même s'ils sont bien meilleurs que dans le cas non hiérarchique, restent bien moins bons que ceux obtenus avec MPICH2 dans le cas homogène. Dans le cas de nœuds hétérogènes, 2.0 GHz et 2.4 GHz, MPICH2 utilise de manière surprenante un algorithme linéaire et par conséquent nos résultats finissent par être meilleurs pour un grand nombre de nœuds, plus précisément aux environs de 120. Cela vient du fait que les algorithmes de ColCOWS restent logarithmiques.

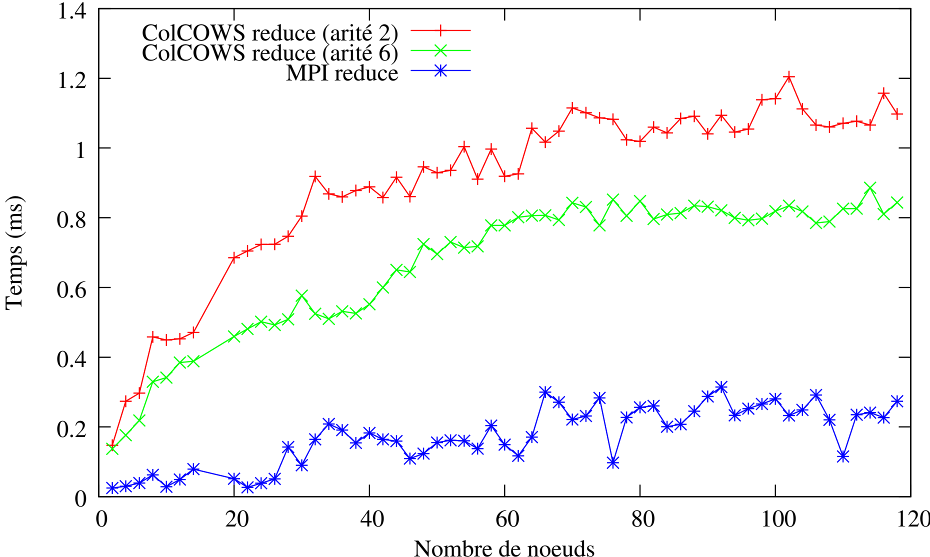


FIGURE 4.6 – Communications collectives : *reduce* pour un calcul de maximum

Dans le cas de communication de type *reduce*, les résultats sont similaires comme le montre la figure 4.6. Les temps sont légèrement meilleurs que pour des *gathers*, ce qui vient du fait que les données transférées sont réduites sur chaque nœud. Le volume de communication est donc plus petit. On peut noter que c'est aussi une arité de 6 qui donne les meilleurs résultats. Ceci s'explique par le fait que les communications effectuées dans le cas d'un *gather* et d'un *reduce* sont les mêmes, la seule différence étant que dans le cas du *reduce* les données sont « *réduites* » par une opération particulière (maximum dans notre cas) à chaque réception intermédiaire.

4.3 Test de validation pour une simulation M-SPMD

Afin d'évaluer les performances de la plate-forme EPSN2 et tout particulièrement nos algorithmes de planification, nous avons développé une simulation de test M-SPMD. Cette simulation est composée de N codes SPMD couplés. Ces codes, tous identiques, sont composés d'une tâche d'initialisation suivie d'une boucle de calcul. Cette boucle est constituée d'une seule tâche que nous appellerons la tâche de calcul. La figure 4.7 illustre la description de la simulation M-SPMD composée du couplage des deux codes SPMD que nous venons de présenter.

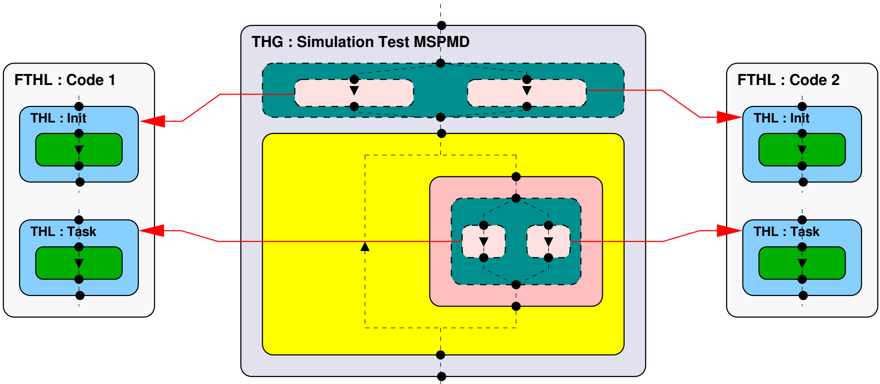


FIGURE 4.7 – Représentation de la simulation M-SPMD de test avec $N = 2$ codes

Pour cette simulation *test*, nous avons plusieurs paramètres pouvant être réglés afin de mesurer leurs influences sur les performances de la plate-forme. Ainsi nous pouvons faire varier le nombre de codes N présents dans le couplage, mais également le nombre de processeurs affectés à chaque code du couplage (P_i pour le code i). Enfin, nous pouvons faire varier le temps tc de la tâche de calcul.

4.3.1 Initialisation de la plate-forme

Tout d'abord, nous avons fait des mesures du temps d'initialisation en fonction du nombre de processus et de codes afin de voir si la plate-forme passe à l'échelle. Nous avons donc dans un premier temps augmenté le nombre de processeurs sur 2 codes de simulation et ce en faisant varier la répartition des processeurs entre ces codes. La figure 4.8 présente ces résultats, chaque courbe correspondant à une répartition du nombre total de processeurs P entre les deux codes.

Comme on peut le constater, cette phase d'initialisation est assez coûteuse. Cela vient essentiellement du fait qu'il faut tout d'abord activer les espaces de travail ColCOWS. Une fois cela fait, il faut lire le fichier des descriptions au format XML et envoyer les différentes informations aux différents *proxies* et aux nœuds. Pendant l'envoi des descriptions, on peut tout de même

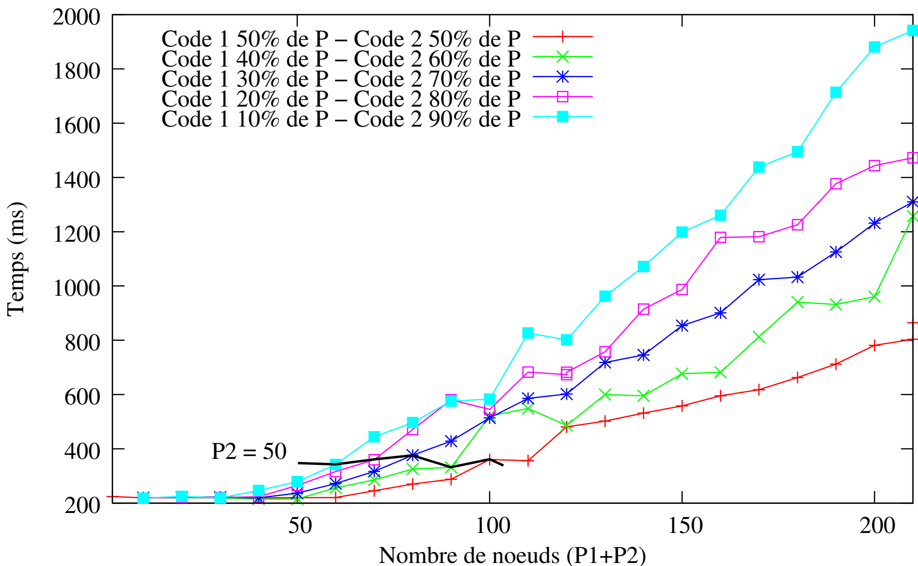


FIGURE 4.8 – Initialisation de la plate-forme : variation du nombre de processeurs sur 2 codes

décrire les objets RedSYM et les enregistrer dans EPSN. Il y a ensuite une phase de synchronisation forte entre tous les nœuds d'un code, suivie d'une synchronisation entre les codes pour s'assurer que tous les composants de la plate-forme sont prêts. Ainsi sur la figure 4.8, on peut noter que les temps d'initialisation sont dictés par le code qui a le plus de nœuds. La courbe noire $P2 = 50$ met ce résultat en avant en indiquant les valeurs du temps d'initialisation pour chaque configuration lorsque P2 s'exécute sur 50 processeurs. Cela s'explique par le fait que jusqu'à environ 30 nœuds, le temps d'initialisation est limité par la création des *proxies* et l'activation du *workspace* des *proxies*. Or pour une simulation donnée, le nombre de proxies est fixe et on a donc un temps constant de plus ou moins 220 *ms*. Au delà des 30 nœuds, c'est l'attente de la fin d'activation du *workspace* des codes qui va être limitant. Cette activation étant faite de façon indépendante sur chaque code, c'est le code possédant le plus de nœuds qui va être le plus lent.

La figure 4.9 correspond à une variation du nombre de codes avec une répartition équilibrée des processus sur tous les codes. Les temps d'initialisation sont quasiment similaires. On a un léger surcoût dû au nombre de codes aux environs de 10 *ms* par code. Ceci vient du temps d'activation du *workspace* contenant les *proxies* des différents codes.

Les temps d'activation de EPSN sont donc assez importants, mais ils n'interviennent qu'une seule fois au cours d'une simulation. Par conséquent, ils n'ont que très peu d'impact sur le temps global des simulations. Il faut tout de même noter que le choix de la technologie CORBA, et surtout l'utilisation du service de nommage, nous limite à ce niveau. En effet, l'activation Col-COWS sollicite fortement ce service. La plate-forme EPSN est donc limitée en nombre maximum de nœuds pouvant être initialisés pour un même code de simulation.

4.3.2 Algorithmes de coordination

Une fois la plate-forme initialisée, nous allons y connecter un client pour qu'il émette des requêtes de pilotage. Cela va nous permettre d'évaluer le temps mis par la plate-forme à planifier la date de traitement et le surcoût que cela engendre sur la simulation. Pour ce faire, nous allons émettre des requêtes un peu particulières, appelées requêtes *test*. Ce sont des requêtes dont l'exécution doit se faire à la même date de point pour tous les nœuds d'un même code.

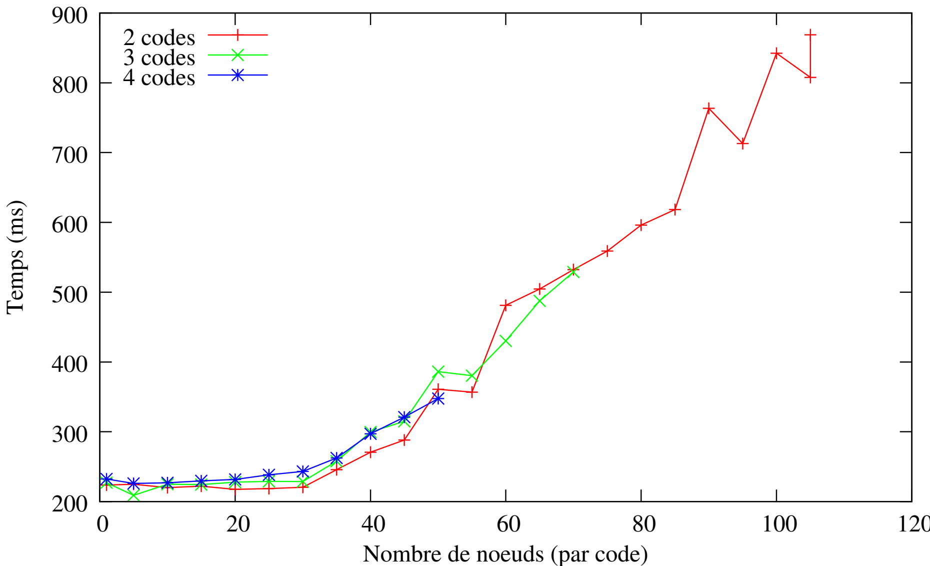


FIGURE 4.9 – Initialisation de la plate-forme : variation du nombre de codes

Une fois la date planifiée atteinte, la requête n'effectue aucun traitement de pilotage et finit instantanément. Cette requête servira essentiellement à mesurer le temps de planification ainsi que le surcoût engendré par la planification sur la simulation distribuée.

Nous avons mesuré la durée moyenne d'une itération de la simulation sans requête, puis nous avons mesuré le surcoût moyen engendré par 500 requêtes exécutées pendant 1000 itérations. Lors de l'émission des requêtes, nous avons mesuré le temps mis par le *master proxy* pour effectuer la planification. Ces mesures ont été faites pour différentes valeurs de la durée tc de la tâche de calcul comprises entre 1 ms et 100 ms.

Il est à noter que pour ces mesures nous nous sommes restreints aux nœuds cadencés à 2.4 GHz du cluster GdX. Nous disposons donc de moins de nœuds, mais les machines sont homogènes et les résultats sont donc plus faciles à analyser.

La figure 4.10 montre les temps de planification interne à la plate-forme EPSN2 lors de la réception d'une requête *test*. Comme nous l'avons vu, l'algorithme de planification est basé sur des communications collectives ColCOWS ; il aura donc une complexité en $O(log(n))$. Suivant le temps de calcul tc, le temps de la planification pour 55 nœuds par code varie entre 8.3 ms et 9.8 ms. Les mesures pour un plus grand nombre de nœuds ont été effectuées et on a des temps compris entre 11 ms et 17 ms pour 110 nœuds par code. Sur le graphique 4.10, on note que lorsque le temps des tâches de calcul est supérieur à 20 ms la durée de la tâche n'influence pas le temps de planification. Mais sur des tâches courtes, il y a un surcoût apparent : environ 1.5 ms de surcoût pour 55 nœuds par code. Ceci vient du fait que plus les tâches de la simulation sont rapides, et plus les chances que les nœuds de la simulation atteignent un point d'instrumentation bloquant sont élevées (étape de « *freeze* » de l'algorithme de planification). Le surcoût vient d'un traitement particulier fait au niveau du *master proxy* lorsque les nœuds notifient le fait qu'ils se sont bloqués.

La figure 4.11 présente le surcoût relatif que les 500 requêtes ont engendré sur les 1000 itérations par rapport au temps « *normal* » de la tâche de calcul. Les résultats montrent bien que les surcoûts ne sont importants que sur les tâches rapides, de l'ordre du temps de planification évalué précédemment. Cela s'explique par le fait que les nœuds atteignent un point d'instrumen-

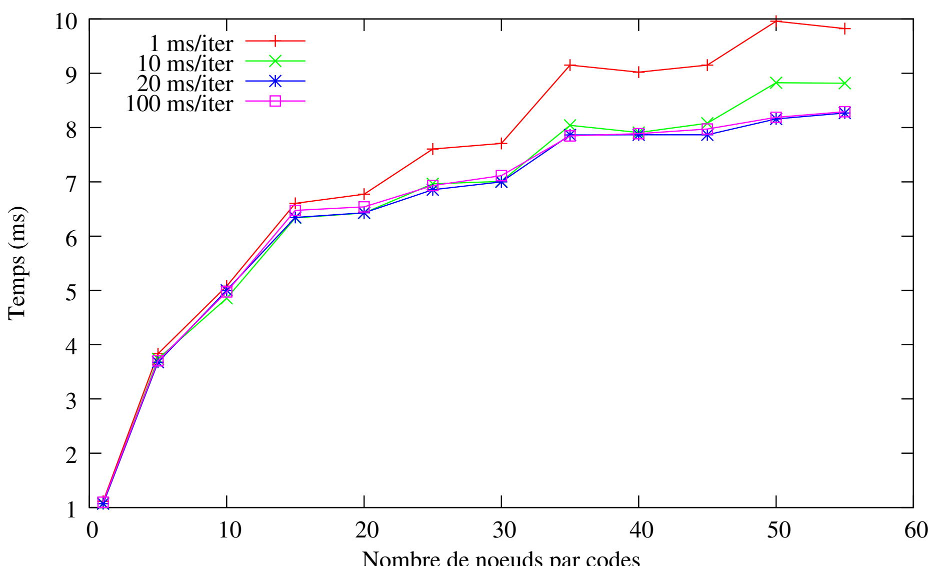


FIGURE 4.10 – Temps de planification pour une requête *test* sur une simulation couplée

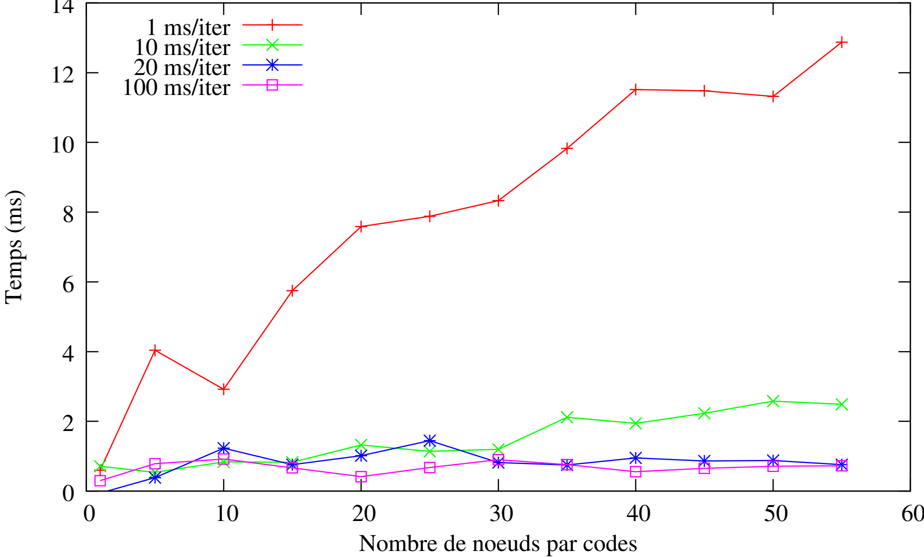


FIGURE 4.11 – Surcoût relatif de la planification d'une requête sur un simulation couplée

tation bloquant la simulation et doivent attendre la fin de la prise de décision pour repartir. À l'inverse, les tâches lentes ne sont que faiblement impactées par la planification car notre algorithme peut être recouvert durant la tâche de calcul. On note uniquement un surcoût inférieur à 1 *ms* lorsque la durée des tâches de calcul dépasse le temps de planification. Une fois que le temps de la tâche de calcul dépasse les 20 *ms*, l'impact de la planification est donc le même. Le léger surcoût vient de l'exécution de la requête. Même si l'exécution de la requête consiste à ne rien faire et à rendre la main, elle fait tout de même intervenir dans EPSN tous les traitements pour l'exécution : vérification de la condition locale, passage dans la file des requêtes exécutables, réveil d'un *worker* et enfin exécution à proprement parler. Une fois la condition locale remplie, la requête doit être exécutée avant que cette condition ne soit plus remplie. Dans notre cas, cela revient à exécuter la requête avant que la date ne change. Or la date planifiée correspond quasiment tout le temps à la date de la fin de la tâche de calcul. Cette tâche est suivie par la fin de la boucle. La requête s'exécute donc entre 2 points d'instrumentation qui se suivent « *instantanément* » et il n'y a donc pas de recouvrement possible.

4.3.3 Client de visualisation parallèle

Pour tester les algorithmes assurant la cohérence des données sur les clients parallèles, nous allons modifier la simulation M-SPMD de test. Pour ce faire, nous lui avons ajouté une tâche de calcul. Nous avons donc deux tâches de calcul, la tâche A et la tâche B. Les temps de calcul de ces deux tâches, tc_A et tc_B, sont réglables. Chaque code contient une donnée de type grille (un tableau 2D distribué en colonne) qui est modifiée dans la tâche A et qui est lisible pendant la tâche B (voir figure 4.12).

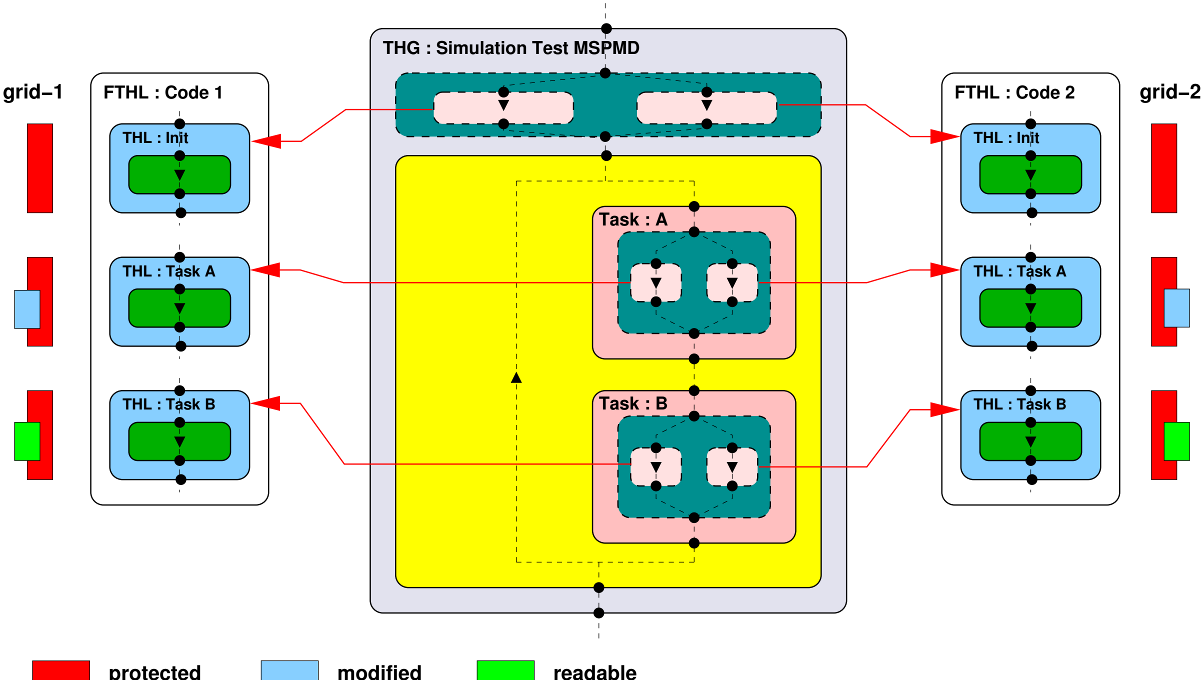


Figure 4.12 – Représentation de la simulation M-SPMD de test pour la visualisation

À cette simulation, nous allons connecter un client parallèle. Ce client va émettre une requête permanente de récupération des données à toutes les itérations. Puis, il va périodiquement vérifier

si une nouvelle version de la donnée a été reçue. Suivant le cas, il va soit utiliser le *lock global* pour protéger les données durant le post-traitement, soit acquitter les données au plus tard (*explicit ack*) après ce post-traitement. Le temps tp du post-traitement est quant à lui réglable. Dans ces mesures, les données sont d'un volume assez faible pour que le temps de transfert n'influe pas sur les surcoûts.

Nous avons effectué différentes mesures pour différentes configurations selon les valeurs de tc_A, tc_B et tp. Le tableau 4.1 récapitule les résultats obtenus pour une exécution de la simulation sur 10 nœuds, soit 5 nœuds par code et 6 nœuds pour le client.

	Configuration (temps en ms)			Mesures (en ms)		Surcoût total
Type acquittement	tc_A	tc_B	tp	tc_A	tc_B	sur $tc_A + tc_B$
Lock Global	1	1	100	1.05	99.08	98.13
Explicit Ack	1	1	100	1.02	101.32	99.34
Lock Global	1	100	100	1.03	100.05	0.08
Explicit Ack	1	100	100	1.02	101.19	1.21
Lock Global	10	1	100	10.05	89.87	88.92
Explicit Ack	10	1	100	10.02	101.35	100.37
Lock Global	100	1	100	99.99	1.37	0.36
Explicit Ack	100	1	100	99.95	101.38	100.33

TABLE 4.1 – Mesures de surcoût dû au post-traitement des données dans un client parallèle

À la vu des résultats, on peut noter que l'algorithme de *lock global* nous donne de bien meilleurs résultats. Dans le cas de l'*explicit ack*, la récupération des données et les post-traitements peuvent être faits uniquement durant la tâche B ; là où les données sont lisibles. Ainsi quel que soit la configuration, les post-traitements n'impactent que la durée de la tâche B, nous notons que le tâche B a un temps mesuré de l'ordre de tp. Une fois le post-traitement de 100 ms effectué, le client doit acquitter la simulation afin de la « *libérer* », ce qui implique un surcoût de l'ordre de 1 ms dans nos expériences. Dans le cas du *lock global*, les post-traitements sont recouverts même durant les tâches où la donnée n'est pas accessible. C'est uniquement au moment où une nouvelle donnée devrait être envoyée que la simulation se bloque. Cela correspond au point d'instrumentation de fin de tâche B de l'itération suivant l'envoi de la donnée. Ainsi la tâche A n'est pas impactée et la tâche B est impactée uniquement si $tc_A + tc_B < tp$. Il faut tout de même noter que les transferts de données, dans le cas où la donnée est verrouillée, peuvent tout de même être effectués selon la configuration de CORBA. Par conséquent, avant la fin du post-traitement, la mémoire relative aux données dans le client peut être au pire doublée.

4.4 Test de validation pour des simulations distribuées réelles

Nous allons à présent voir comment la plate-forme se comporte sur un exemple de pilotage de simulation réelle. Nous allons également voir comment nous avons fait pour rendre ces simulations pilotables. Pour cela, nous allons étudier deux simulations. Ces simulations sont les simulations que nous avons pris comme exemples dans l'état de l'art, LibMultiScale et DLPoly/Siesta. Nous allons essentiellement détailler l'exemple de LibMultiScale car c'est un code M-SPMD. Le couplage DLPoly/Siesta est de type Client/Serveur et ce type de couplage n'est pas encore totalement pris en compte par la plate-forme EPSN2.

4.4.1 LibMultiScale : simulation couplée de la dynamique moléculaire avec les éléménts finis

La plate-forme LibMultiScale, comme nous l'avons déjà vu, est une plate-forme de couplage de codes multi-échelles. Les codes couplés sont un code d'éléments finis développé par le Laboratoire de Simulation de la Mécanique des Solides (EPFL-ENAC-IIS-LSMS) et un code de dynamique moléculaire LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [81] développé au Sandia National Labs. Ces codes sont tous deux des codes parallèles utilisant la bibliothèque MPI pour leurs communications internes. LAMMPS est développé en C++ et le code de mécanique des solides est quant à lui développé en C. Par la suite, on notera MD le code de dynamique moléculaire et FE le code d'éléments finis.

La simulation débute par une phase d'initialisation, suivie de la boucle principale de calcul. Durant la phase d'initialisation, les deux codes lisent les données des modèles et construisent les structures de données nécéssaires (maillage non structuré pour FE et réseau de points pour MD). Le couplage en lui-même nécessite une phase d'initialisation commune aux deux codes. La boucle principale qui suit peut être également décomposée en plusieurs étapes. Tout d'abord, on a des étapes de calcul effectuées en parallèle par les deux codes de simulation. Ces étapes sont, la mise à jour des positions (*updatepositions*), le calcul des différentes forces (*updateforces*) qui s'appliquent sur les degrés de liberté et la mise à jour des vitesses (*updatevelocities*). Finalement, nous avons l'étape de couplage à proprement parler (*updatecoupling*) qui est effectuée de manière distribuée sur les deux codes.

Afin de piloter cette simulation, nous l'avons donc instrumenté conformément à la description ci-dessus. Même si chaque étape de calcul pourrait être raffinée du point de vue de la modélisation dans EPSN, un tel raffinement n'apporterait rien *a priori* pour le pilotage de cette simulation. La figure 4.13 est une capture d'écran du client Simone connecté à la simulation instrumentée. Cette capture d'écran présente une vue graphique de la THG décrivant une simulation type couplée par la LibMultiScale. Les annexes A.4, A.5, A.6 et A.7 présentent des listings correspondant à la description en XML de la THG de la simulation, ainsi que des extraits simplifiés de code-source illustrant l'instrumentation du code.

Avant de faire des mesures de performance de la plate-forme EPSN2 sur cette simulation, nous pouvons noter que le code du coupleur LibMultiScale (qui est le code qui a été instrumenté) fait approximativement 61 000 lignes. Or l'instrumentation avec l'API EPSN fait un peu plus de 500 lignes (ce qui représente moins de 1% du code) dont l'essentiel (80%) concerne la description RedSYM des données distribuées.

Par la suite, nous allons utiliser deux cas-tests pour faire nos mesures sur cette simulation. Le premier est un cas-test 2D de petite taille, le deuxième un cas-test 3D beaucoup plus volumineux. Sur ces deux tests nous allons mesurer l'impact du pilotage via la plate-forme EPSN. Nous pourrons alors comparer nos résultats à la méthode de post-traitement native à LibMultiScale de sauvegarde dans des fichiers sur disque. Concrètement, cette méthode permet de sauvegarder des données dans le format de fichier de VTK avec une période (en nombre d'itérations) ajustable. La classe C++ en charge de cette fonctionnalité est nommée *dumper* Paraview. Afin de respecter cette convention de nom, nous avons donc ajouté une classe *dumper* EPSN qui va initialiser la plate-forme et décrire les données que l'on souhaite visualiser au travers du pilotage.

4.4.1.1 Cas-test 2D de propagation d'onde dans un crystal d'Argon

Le cas test 2D correspond à une stimulation d'un crystal d'Argon. Une stimulation, sous la forme de déplacement gaussien, est initialement imposée au matériau. Dès lors, une onde

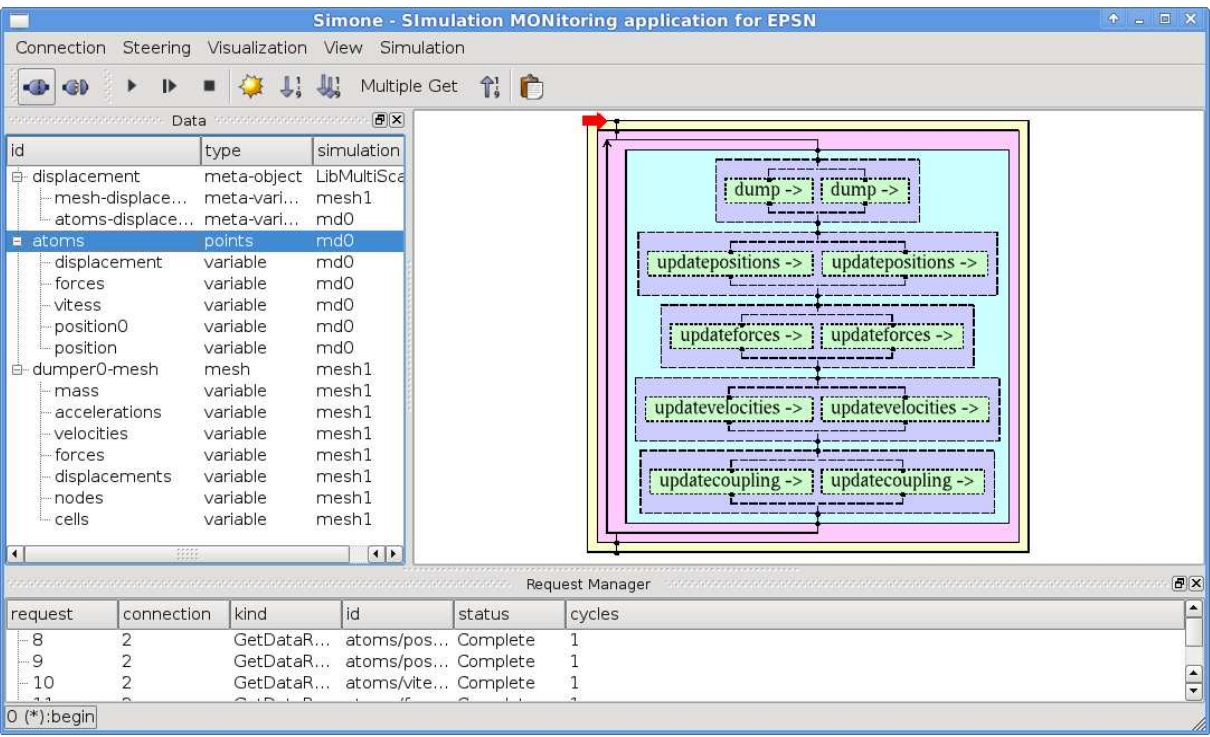


FIGURE 4.13 – Client Simone connecté à la simulation LibMultiScale

se crée et est amenée à traverser la zone de couplage. Il s'agit ici d'un cas test très classique dans la caractérisation des méthodes de couplage qui permet d'évaluer le taux de réflexion d'ondes à l'interface de couplage. A l'aide de la LibMultiScale, on peut alors tester le couplage via différents codes de simulation. Nous l'avons donc utilisé pour tester l'instrumentation du coupleur. Le cas-test est composé de 286 262 atomes pour la partie MD, d'un maillage constitué de 26 130 éléments triangulaires et de 13 347 sommets pour la partie FE.

Nous avons tout d'abord mesuré le temps de référence qui est le temps moyen d'un pas de temps sans instrumentation et sans écriture de données sur disque. Le cas est exécuté sur 8 processeurs, 4 pour MD et 4 pour FE et il met en moyenne 111.76 ms par itération. Nous avons ensuite mesuré l'impact de l'instrumentation avec EPSN sans aucune interaction de pilotage. Le temps moyen de 112.83 ms est obtenu, soit un surcoût de 0.9 %.

	Surcoût			
Période (nb d'itérations)	1		10	
Dumper Paraview	531.16 ms	375.3 %	154.05 ms	37.8 %
Dumper EPSN (Explicit Ack)	284.39 ms	154.5 %	124.18 ms	11.1 %
Dumper EPSN (Lock Global)	274.32 ms	145.5 %	124.90 ms	11.8 %

TABLE 4.2 – Mesure des surcoûts du post-traitement (sauvegarde/transfert)

La table 4.2 présente les surcoûts obtenus lors de l'utilisation du dumper Paraview et du dumper EPSN avec uniquement des transferts vers un client. Pour ce faire, nous avons comparé une écriture de fichier à un *get* depuis un client EPSN séquentiel distant, toutes les itérations et toutes les 10 itérations. On note que la récupération des données via la plate-forme EPSN

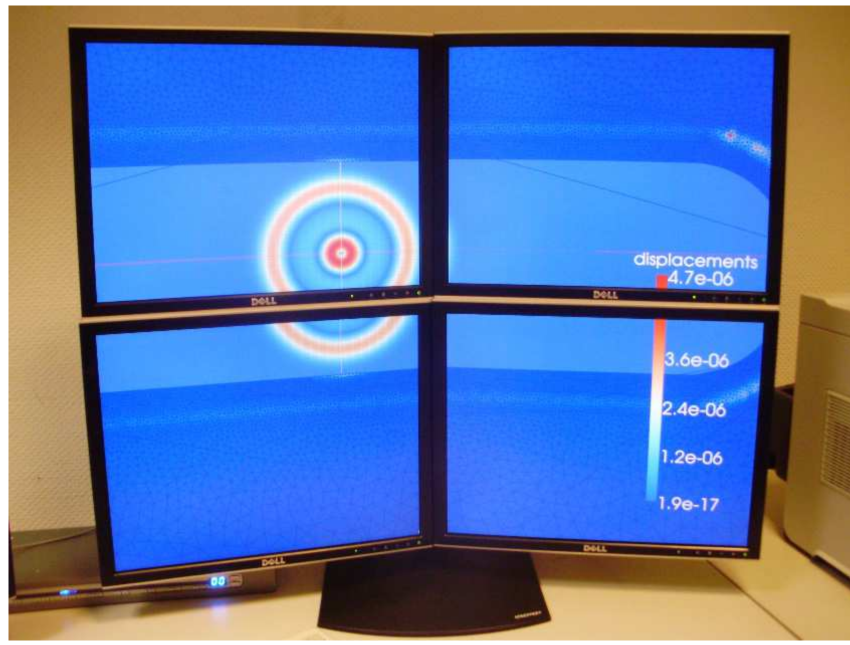


FIGURE 4.14 – Visualisation des résultats du cas 2D de LibMultiScale sur mur d'écran

est bien moins coûteuse. À ce niveau, la différence entre les méthodes d'*explicit ack* et de *lock global* n'est pas notable car nous n'effectuons pas de post-traitement pour la visualisation. Par conséquent, l'acquittement intervient quasiment au même moment dans les deux cas.

	Surcoût			
Période (nb d'itérations)	1		10	
En séquentiel (Explicit Ack)	450.82 ms	303.4 %	147.66 ms	32.12 %
En séquentiel (Lock Global)	308.65 ms	176.2 %	125.06 ms	11.9 %
En parallèle sur 4 nœuds (Lock Global)	242.08 ms	116.6 %	114.09 ms	2.1 %

TABLE 4.3 – Mesure des surcoûts du post-traitement (visualisation) pour le client EPSN distant séquentiel et parallèle

Par la suite, nous avons fait nos mesures en rajoutant un post-traitement classique de visualisation afin d'afficher le déplacement des atomes (voir figure 4.14). La table 4.3 présente donc les résultats obtenus avec la plate-forme EPSN pour un client distant séquentiel ou parallèle. On peut noter que la méthode du *lock global* donne de bien meilleurs résultats, comme on pouvait s'y attendre. Cela est dû au fait que les post-traitements peuvent être recouverts sur une itération complète et non pas uniquement sur les plages d'accès aux données. Dans le cas d'une requête *get* permanente récupérant les données toutes les 10 itérations, on a alors un surcoût comparable au surcoût de transfert seul (table 4.2, 124.90 ms), car on peut recouvrir les post-traitements sur les 10 itérations qui séparent deux transferts. Les gains dûs à la visualisation en parallèle sont essentiellement dûs au fait qu'avec un client parallèle nous pouvons agréger plusieurs flux réseau. Dans notre cas, nous pouvons avoir jusqu'à 4 flux Giga-Ethernet agrégés sur le lien 10G Ethernet reliant le cluster de calcul au cluster graphique. En revanche, l'apport du rendu parallèle n'est pas très important car le volume des données à traiter est insuffisant.

4.4.1.2 Cas test 3D de contact glissant de surfaces rugueuses

Le second cas test est un cas 3D mettant en jeu un volume de données à traiter bien plus important. Deux objets présentant des surfaces rugueuses sont mis en contact par l'application d'une pression. Une fois l'équilibre obtenu, un glissement par contrôle du champ de déplacement est imposé de sorte que les deux objets glissent l'un sur l'autre. La mécanique continue avec une formulation en éléments finis permet de modéliser les champs de déformation élastique lointains, tandis que la zone de contact qui va observer des déformations importantes est modélisée à l'aide de la dynamique moléculaire. Cette approche permet de réduire le coût de calcul tout en maintenant une précision suffisante pour calculer les déformations surfaciques et les forces de frictions. Du fait de la nature tri-dimentionnelle de cette simulation, le calcul est beaucoup plus coûteux que le précédent cas 2D. Plus précisément, ce cas test nécessite 1 033 124 atomes et un maillage constitué de 41 472 éléments tétraèdriques et de 7 681 sommets.

Nous avons exécuté ce cas test sur 50 processeurs, répartis en 40 processeurs pour MD et 10 processeurs pour FE, et nous avons obtenu un temps moyen pour une itération de 469.05 ms. Dans le cas de l'instrumentation avec EPSN (sans interaction), ce temps passe à 471.31 ms soit un surcoût de 0.5 %.

	Surcoût			
Période (nb d'itérations)	1		10	
Dumper Paraview	1 220.70 ms	160.2 %	542.93 ms	15.8 %
Transfert EPSN en séquentiel (Explicit Ack)	888.10 ms	89.3 %	519.97 ms	10.9 %
Transfert EPSN en séquentiel (Lock Global)	864.79 ms	84.4 %	516.10 ms	10.0 %
Transfert EPSN en parallèle (Explicit Ack)	698.77 ms	49.0 %	494.14 ms	5.3 %
Transfert EPSN en parallèle (Lock Global)	687.58 ms	46.5 %	490.26 ms	4.5 %

Table 4.4 – Mesure des surcoûts du post-traitement (sauvegarde/transfert)

Comme nous l'avons déjà dit, sur ce cas test, le volume de données est bien plus important que précédemment : au total, 23, 64 Mo par variable pour les atomes contre 6.55 Mo pour le cas 2D. Par conséquent, les temps de transfert sont plus importants, mais nous disposons également de plus de temps pour recouvrir ces transferts. Le surcoût relatif est donc moins important que dans le cas 2D comme nous pouvons le voir dans la tableau 4.4. Dans le cas de transferts parallèles des 10 nœuds du cluster de calcul Borderline vers 4 des nœuds du cluster graphique, on a un gain assez important (environ 180 ms pour une période de 1 itération). Ceci est dû, comme dans le cas 2D, au fait que l'on agrège les flux parallèles sur le lien 10G Ethernet. Dans tous les cas, la différence est minime si on compare la méthode de l'*explicit ack* à celle du *lock global*, car dans les deux cas l'acquittement intervient quasiment au même moment.

	Surcoût			
Période (nb d'itérations)	1		10	
Séquentielle (Explicit Ack)	2 857.14 ms	509.1 %	756.91 ms	61.4 %
Séquentielle (Lock Global)	1 200.73 ms	156.0 %	516.98 ms	10.2 %
Parallèle (Explicit Ack)	1 631.56 ms	247.8 %	623.19 ms	32.9 %
Parallèle (Lock Global)	761.40 ms	62.3 %	494.20 ms	5.3 %

Table 4.5 – Mesure des surcoûts du post-traitement de visualisation pour un client EPSN séquentiel ou parallèle

Le tableau 4.5 présente les temps obtenus pour un client de visualisation pouvant être séquentiel ou parallèle. Nous constatons que la visualisation séquentielle engendre un surcoût bien plus important que dans le cas 2D. Dans le meilleur des cas, on arrive avec le *lock global* à des temps comparables à ceux obtenus avec le *dumper* Paraview (table 4.4, 1 220.70 ms). Par contre, dans le cas d'un client parallèle, les surcoûts sont nettement plus faibles. Si l'on compare les temps de transfert séquentiel et parallèle nous gagnons environ 180 ms, alors que les gains dans le cas de la visualisation parallèle sont de 1 220 ms (table 4.5, 2 857.14 ms contre 1 631.56 ms) pour l'*explicit ack* et 440 ms (table 4.5,1 200.73 ms contre 761.40 ms) pour le *lock global.* Le gain du parallélisme dans ce cas ne vient donc pas uniquement du transfert, il vient également du fait que le traitement des données est réparti. Par conséquent, malgré le coût de rendu parallèle un peu plus élevé qu'en séquentiel, le temps de post-traitement est globalement plus faible en parallèle.

Dans le cas de la visualisation parallèle avec une requête *get* de période 10 itérations, nous arrivons donc à recouvrir intégralement les temps de visualisation et nous nous ramenons au surcoût dû uniquement au transfert.

Il est à noter que nous avons fait nos mesures avec une période de 1 itération et de 10 itérations entre les transferts, ce qui n'est pas très réaliste sur ce cas test car sur 10 itérations les évolutions physiques ne sont pas très visibles. Mais nous avons fait cela afin de mieux voir l'impact de notre méthode. Si nous avions fait un *get* permanent avec une période de 100 itérations (ce qui est la période utilisée par défaut avec le *dumper* Paraview), nous aurions eu une itération moyenne de 471.23 ms, soit un surcoût comparable au surcoût de l'instrumentation (0.5 %). La plate-forme EPSN permet donc d'effectuer une visualisation « *en-ligne* » en impactant que faiblement la simulation.

4.4.2 Couplage DLPoly/Siesta

Nous allons à présent décrire le couplage DLPoly/Siesta plus en détails. Nous allons uniquement faire la description en tâches hiérarchiques de ce couplage car la plate-forme EPSN ne prend pas complètement en compte les simulations de type Client/Serveur. Le couplage DLPoly/Siesta est un couplage QM/MM (Quantum Mechanics/Molecular Mechanics).

DLPoly, le code de dynamique moléculaire [32], est composé d'une boucle en temps au cours de laquelle il va calculer les forces qui s'appliquent sur différentes molécules. Puis, il va mettre à jour les positions des molécules et les vitesses.

Le code de mécanique quantique, Siesta [127], va quant à lui calculer les forces des atomes sur la partie quantique ainsi que la densité de charge à partir de la position des atomes et du champ électrique extérieur au domaine.

Le couplage de ces codes, effectué dans le cadre de l'ANR NOSSI [102], distingue deux zones de l'espace : une zone quantique (zone QM) et une zone de dynamique moléculaire (zone MM). À chaque itération, le code de dynamique moléculaire va calculer les forces des molécules de la zone MM et avec la force calculée par le code de mécanique quantique il va pouvoir déduire les nouvelles positions des molécules dans les deux zones. Il va ensuite envoyer les nouvelles positions au code QM. Dans ce couplage, un troisième code intervient potentiellement pour extraire des propriétés optiques sur les matériaux modélisés. Périodiquement, le code de mécanique quantique va faire appel à un code de TD-DFT afin de calculer le spectre d'absorption du matériau. Cela permet aux scientifiques de confronter la simulation à des expériences réelles. La figure 4.15 montre une vue d'ensemble du couplage de ces codes.

A partir de cette description de haut-niveau des codes nous allons pouvoir définir la THG et les THL décrivant ces simulations pour la plate-forme EPSN. Cela nous donnera l'ensemble des

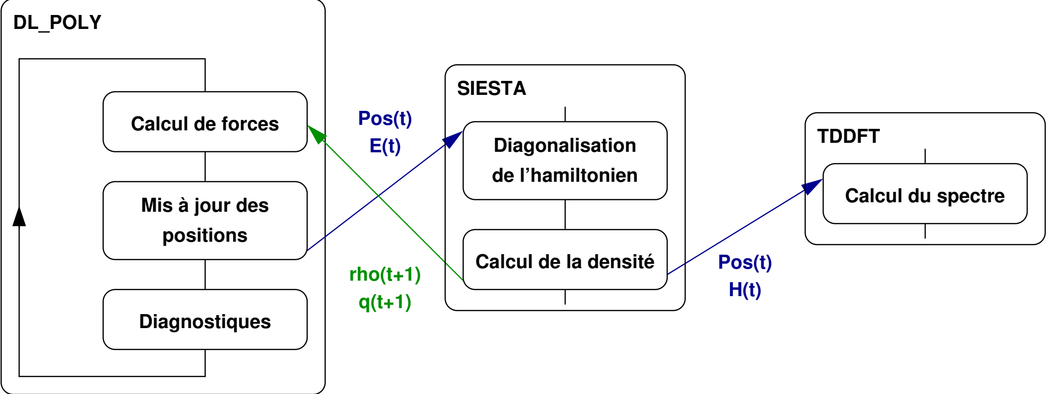


FIGURE 4.15 – Vue d'ensemble du couplage DLPoly/Siesta

points d'instrumentation à intégrer dans les codes-sources. Le listing 4.1 présente la description en langage XML de cette simulation couplée. Dans cette simulation le code DLPoly joue le rôle de client et fait un appel de méthode sur le code serveur Siesta (voir ligne 32 du listing 4.1). Le code Siesta, en plus d'être un code serveur pour DLPoly, est un client pour le code de TDDFT (voir ligne 58 du listing 4.1). Dans ce couplage, les données qui nous intéresse pour la visualisation sont les positions des atomes et leurs charges pour la partie MM, ainsi que la densité électronique pour la partie QM. Ces informations sont donc représentées par la méta-donnée *qm_mm* décrit à la ligne 11 du fichier de description.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2
3  <epsn>
4    <simulation id="dl_poly">
5      <data>
6        <object class="particles" id="atoms">
7          <variables id="position" />
8          <variables id="Q" label="charge" />
9        </object>
10
11       <meta-object id="qm_mm">
12         <meta-variable id="dm-position" data_id="atoms"
13                        variable_id="position"
14                        simulation_id="dl_poly" />
15         <meta-variable id="dm-q" data_id="atoms"
16                        variable_id="Q"
17                        simulation_id="dl_poly" />
18         <meta-variable id="qm" data_id="qm"
19                        variable_id="rho"
20                        simulation_id="siesta" />
21       </meta-object>
22     </data>
23
24     <mht auto-start="false">
25       <task id="init" />
26
27       <loop id="mainloop">
28         <task id="body">
29           <concurrent id="compute_force">
30             <task id="local" />
31             <remote id="remote" task_id="qm_forces"
32                     simulation_id="siesta" />
33             <task id="integration" />
34           </concurrent>
```

```
35
36          <task id="compute_position" />
37
38          <task id="diagnostic" />
39        </task>
40      </loop>
41    </mht>
42  </simulation>
43
44 <!-- ============================================================== -->
45  <simulation id="siesta">
46    <data>
47      <object class="grid" id="qm">
48        <variable id="rho" label="densite␣electronique" />
49      </object>
50    </data>
51
52    <lht id="qm_forces">
53      <task id="init" />
54      <task id="compute" />
55      <switch id="compute_spectre">
56        <task id="no-spectre" />
57        <remote id="spectre" task__id="compute_tddft"
58                simulation__id="tddft" />
59      </switch>
60    </lht>
61  </simulation>
62
63 <!-- ============================================================== -->
64
65  <simulation id="tddft">
66    <data>
67      <object class="grid" id="spectre" label="spectre␣d'absoption">
68        <variable id"I" label="intensite" />
69      </object>
70    </data>
71
72    <lht id="compute_tddft">
73      <task id="init" />
74      <task id="compute" />
75      <task id="save" />
76    </lht>
77  </simulation>
78 </epsn>
```

Listing 4.1 – Description en XML de la THG de la simulation couplée DLPoly/Siesta

4.5 Conclusion

Ces résultats mettent en évidence les performances de la plate-forme EPSN2, mais également la pertinence du pilotage pour des simulations distribuées. En effet, nous arrivons à obtenir une visualisation *en ligne* en perturbant moins la simulation que par des techniques de post-traitement plus classiques. Notre approche dépend tout de même de trois facteurs : le temps moyen d'une itération de la simulation, le temps de transfert des données que l'on souhaite visualiser et le temps du post-traitement. Idéalement, on souhaite que la simulation s'exécute suffisamment rapidement pour que le pilotage puisse être interactif. Pour ce faire, il est possible d'ajuster le nombre de processeurs de la simulation (sous réserve que le code ait une bonne scalabilité). Ceci étant posé, pour visualiser les données *en ligne*, il faut pouvoir dimensionner le réseau et les machines de *post-traitement* de telle sorte que le surcoût induit par le pilotage ne soit pas trop important. De plus, il faut pouvoir déterminer pour une simulation donnée la période

idéale pour récupérer les données ; cette période doit être assez petite pour permettre de suivre l'évolution du phénomène physique, mais également assez grande pour que les post-traitements puissent être recouverts le plus possible.

Conclusion et perspectives

En conclusion de cette thèse, nous allons donner des perspectives pour nos différentes contributions : (1) le modèle de pilotage pour les simulations distribuées et (2) la réalisation de la plate-forme EPSN2 intégrant ce modèle.

Modèle de pilotage de simulations distribuées

Ces travaux consistaient dans un premier temps à concevoir une plate-forme de pilotage pour les simulations distribuées. Pour ce faire, nous avons étendu le modèle de pilotage introduit par A. Esnard dans [40]. Le MHT parallèle atteint ses limites lorsqu'il s'agit de piloter des simulations distribuées autres que SPMD. Nous l'avons donc étendu au MHT distribué en proposant une extension de la description en tâches hiérarchiques et au système de dates, ce qui a nécessité de nouveaux algorithmes de planification. Afin d'assurer la cohérence des traitements de pilotage, nous avons également dû modifier la description des données et ce par l'ajout des méta-données. Notre modèle se limite tout de même aux simulations de types M-SPMD et Client/Serveur. Nous nous sommes restreints à ces types de simulations car ce sont les plus répandues dans le cadre des simulations HPC.

Néanmoins, il existe d'autres types de simulations distribuées que nous ne gérons pas aujourd'hui, comme par exemple les simulations paramétriques. Ce sont des simulations généralement de type maître/esclaves où les esclaves exécutent tous la même tâche sur un jeu de paramètres différents. Ces simulations servent par exemple à évaluer l'impact des différents paramètres sur un certain phénomène physique (étude de sensibilité). En perspective, on peut envisager différents types de pilotage pour ces simulations.

Dans un premier temps, on peut envisager un pilotage au niveau du maître. Cela consisterait à piloter un code séquentiel ou SPMD. Ce type de pilotage revient essentiellement à contrôler les paramètres envoyés aux différents esclaves et potentiellement à injecter de nouveaux paramètres à tester à la volée. Ce type de pilotage est tout à fait réalisable avec la version parallèle du modèle.

Le second type de pilotage consisterait à contrôler les différents esclaves. Dans ce cas, nous avons plusieurs codes identiques (séquentiels ou parallèles) qui s'exécutent simultanément. Ces codes peuvent être vus comme autant de codes indépendants, mais la cohérence des interactions ne peut être que locale aux codes. Si nous voulons par exemple comparer des informations issues de ces différents codes, il faut pouvoir s'assurer qu'elles sont cohérentes entre elles. Pour ce faire, nous ne pouvons pas nous limiter à une vision simple des codes, car il faut un moyen de décrire ce qui relie ces informations entre elles. Plus précisément, il faut définir en quoi consiste la cohérence de telles informations distribuées. Cela peut être fait à l'aide des méta-données, la difficulté résidant dans le fait que le nombre de code n'est pas à priori connu lors de la description. Il faudrait donc une version dynamique des méta-données.

Par ailleurs, nous pourrions enrichir les contextes des tâches afin de proposer de nouvelles actions de pilotage ou d'optimiser les actions déjà existantes. Nous pourrions par exemple ajouter des informations concernant les tâches de la simulation effectuant des transferts de données en interne. Ainsi nous pourrions, dans la mesure du possible, éviter d'effectuer des transferts de données pour le pilotage au cours de ces tâches afin de perturber le moins possible la simulation. Nous pourrions également fournir des outils tel que du *checkpointing*. Pour ce faire, il faudrait introduire un contexte indiquant que l'on peut sauvegarder les données nécessaires au mécanisme de reprise de la simulation. Cela nécessite de décrire l'ensemble des données utiles au *checkpointing* en nous appuyant sur le modèle de données de EPSN. Après quoi, ces données pourraient être sauvegardées via des I/O asynchrones dans un format standard (*e.g.* HDF5) ou transférées via EPSN vers une base de données distante. Par la suite, pour pouvoir reprendre l'exécution de la simulation sur un point de reprise, un client EPSN pourrait envoyer les données préalablement sauvegardées avant de lancer la reprise effective des calculs ; cela devrait être fait au moment du redémarrage la simulation.

À la vue des résultats, nous avons démontré que la plate-forme EPSN2 pouvait piloter des simulations possédant plus d'une centaine de nœuds. Ceci a été rendu possible essentiellement par l'introduction de communications hiérarchiques. Ces résultats sont encourageants, mais ne permettent pas encore le pilotage de simulations à très grande échelle, car les coordinateurs (*proxies*) représentent un goulot d'étranglement dans certaines étapes des algorithmes. Pour corriger cela, il faudrait une version totalement distribuée des algorithmes, en distribuant notamment les prises de décision sur les nœuds selon la même hiérarchie déjà utilisée pour les communications collectives.

Réalisation d'une plate-forme de pilotage

Les résultats présentés au chapitre 4, mettent en avant le fait que le prototype que nous avons développé se comporte bien mais a tout de même certaines limitations. En effet, dans certaines phases telles que l'initialisation ou la synchronisation des nœuds, nous avons un schéma de communication de type N vers 1. Cela est limitant lorsque le nombre de nœuds devient grand. La première phase limitée par ce type de communication est la phase d'activation des *workspaces* ColCOWS. Durant cette étape, le service de nommage CORBA est limitant. Comme nous avons fait le choix de ne pas modifier la façon dont la simulation est déployée, nous devrons donc modifier la façon dont nous activons les nœuds de la plate-forme EPSN associés aux processus de la simulation. Une solution serait de réécrire un service de nommage distribué à base d'une technologie pair-à-pair par exemple afin de répartir les connexions sur plusieurs processus formant un même service de nommage.

La seconde phase où nous avons des communications de type N vers 1 réside dans les étapes où les nœuds doivent signaler aux *proxies* la fin d'un traitement. Cela intervient lors des synchronisations afin de détecter la terminaison globale de l'algorithme. Pour améliorer cela, il faudrait modifier la librairie ColCOWS afin de pouvoir hiérarchiser la remontée d'informations vers les *proxies*. De plus, nous pourrions améliorer les prises de décision de planification en prenant des décisions localement sur des sous-arbres, ce qui irait dans le sens de l'amélioration des algorithmes de planification afin qu'ils soient plus distribués.

Il faudrait également rendre ColCOWS plus dynamique ; pour le moment, une fois un *workspace* activé, le nombre de nœuds formant ce *workspace* est figé. Le fait de pouvoir ajouter ou enlever dynamiquement des nœuds permettrait d'avoir plus de liberté dans la plate-forme EPSN.

Par exemple, cela permettrait de lever une des limitations actuelles pour l'intégration des simulations Client/Serveur dans EPSN. En effet pour ces simulations, le client et les serveurs doivent actuellement être initialisés simultanément au démarrage de la plate-forme. Les serveurs sont donc fixes tout au long de la simulation et on ne peut pas en rajouter ou en enlever dynamiquement.

Par ailleurs, compte tenu de l'évolution actuelle des codes de simulations vers du parallélisme hybride, il serait intéressant d'étendre la plate-forme afin de prendre en charge de tels codes de simulations. On peut noter que pour des simulations hybrides de type multi-processus/multi-thread, le découpage en tâches ne change *a priori* pas par rapport à des simulations parallèles classiques et notre modélisation reste *a priori* valable sous réserve de bien respecter un paradigme SPMD. Pour le moment, pour pouvoir piloter de telles simulations, il faut que les tâches soient assez grossières pour englober les sous-tâches qui sont reparties entre threads. Par exemple, dans le cas de threads OpenMP, cela revient donc à instrumenter le code autour des régions *parallèles*.

Enfin, la plate-forme EPSN pourrait fournir via les mécanismes de pilotage, des outils d'aide au développement. Nous avons déjà parlé des possibilités de faire du *checkpointing*, mais nous pourrions également fournir des outils de *profiling* comparable à TAU [131]. En effet, la plate-forme peut déjà en interne générer des traces d'exécution afin d'aider au développement des différents algorithmes. Ces traces pourraient être remontées au niveau de l'API utilisateurs afin que les développeurs des codes de simulation puissent l'utiliser pour afficher en temps réel dans un client de pilotage l'enchaînement des tâches du code, le temps passé dans chaque tâche, les communications entre les processus et les changements d'états. Cela permettrait de faire une analyse visuelle du temps passé dans les tâches et des schémas de communications.

La plate-forme EPSN2

Les développements sur la plate-forme EPSN2 ont été réalisés au sein du projet ScAlApplix [5] puis du projet HiePACS [6] de l'INRIA Bordeaux – Sud-Ouest et dans le cadre de l'ANR MASSIM [7]. Ces développements ont été faits en collaboration avec M. Souchaud pour la bibliothèque ColCOWS et G. Caramel pour le client EPSN, Simone. La bibliothèque EPSN2 est disponible sur la GForge de l'INRIA sous licence CeCILL-C (`http://epsn.gforge.inria.fr`).

5. ScAlApplix : Schémas et algorithmes hautes performances pour les applications scientifiques complexes.
6. HiePACS : High-End Parallel Algorithms for Challenging Numerical Simulations.
7. MASSIM : Développement d'un environnement logiciel pour le traitement et la visualisation interactive de MASses de données complexes de grande taille issues de la SIMulation numérique (ANR-05-MMSA-0008-03).

Annexe A

API de EPSN

A.1 Descritption XML complète de la simulation Δ

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<epsn>
  <simulation id="delta">
    <data>
      <meta-object id="values">
        <meta-variable id="values-0" data_id="Data0"
                       variable_id="values"
                       task_id="procedure00"
                       simulation_id="delta1"
                       mask="0.1.1t.0.1.3t" />
        <meta-variable id="values-1" data_id="Data1"
                       variable_id="position"
                       task_id="procedure10"
                       simulation_id="delta2"
                       mask="0.1.1t.0.1.2t" />
        <meta-variable id="values-1" data_id="Data1"
                       variable_id="values"
                       task_id="procedure10"
                       simulation_id="delta2"
                       mask="0.1.1t.0.1.2t" />
      </meta-object>
    </data>

    <ght auto-start="false">
      <task id="begin" />

      <loop id="mainloop">
        <task id="mainbody">
          <remote id="remote0" task_id="procedure00"
                  simulation_id="delta1" />

          <remote id="remote1" task-id="procedure10"
                  simulation_id="delta2" />

          <concurrent id="concurrent">
            <remote id="remote0" task_id="procedure01"
                    simulation_id="delta1" />
            <remote id="remote1" task_id="procedure11"
                    simulation_id="delta2" />
          </concurrent>
        </task>
      </loop>

      <task id="end" />
    </ght>
```

```
47   </simulation>
```

Listing A.2 – Description XML correspondant à la simulation Δ (Description du code client).

```
1    <simulation id="delta1">
2      <data>
3        <object class="grid" id="Data0">
4          <variable id="values" />
5        </object>
6      </data>
7
8      <lht id="procedure00">
9        <data_context ref="Data0" context="readable"/>
10       <switch id="switch">
11         <data_context ref="Data0" context="protected"/>
12         <task id="switch0" />
13         <task id="switch1" />
14       </switch>
15
16       <loop id="subloop0">
17         <task id="subbody0">
18           <data_context ref="Data0" context="modified"/>
19         </task>
20       </loop>
21
22       <point id="point" />
23     </lht>
24
25     <lht id="procedure01">
26       <task id="task0" />
27       <task id="task1" />
28     </lht>
29   </simulation>
30
31
32   <simulation id="delta2">
33     <data>
34       <object class="points" id="Data1">
35         <variable id="position" />
36         <variable id="values" />
37       </object>
38     </data>
39
40     <lht id="procedure10">
41       <data_context ref="Data0" context="protected"/>
42       <task id="begin">
43         <data_context ref="Data0" context="writable"/>
44       </task>
45
46       <loop id="subloop1">
47         <data_context ref="Data0" context="readable"/>
48         <task id="subbody1">
49           <data_context ref="Data0" context="modified"/>
50         </task>
51       </loop>
52
53       <task id="end" />
54     </lht>
55
56     <lht id="procedure11">
57       <task id="task0" />
58       <task id="task1" />
59     </lht>
60   </simulation>
61 </epsn>
```

Listing A.3 – Description XML correspondant à la simulation Δ (Description des serveurs).

A.2 Différents listings lié a la simulation LibMultiScale

```
<epsn>
  <simulation id="mesh1">
    <data>
      <object class="mesh" id="dumper0-mesh">
        <variable id="cells"/>
        <variable id="nodes"/>
        <variable id="displacements"/>
      </object>
    </data>

    <htm auto-start="false" >
      <data_context ref="dumper0-mesh" context="readable"/>
      <loop id="main">
        <task id="body">
          <task id="dump" ></task>
          <task id="updatepositions">
            <data_context ref="dumper0-mesh"
                          variable-id="displacements"
                          context="modified"/>
          </task>
          <task id="updateforces"></task>
          <task id="updatevelocitie"></task>

          <task id="updatecoupling">
            <data_context ref="dumper0-mesh"
                          variable-id="displacements"
                          context="modified"/>
          </task>
        </task>
      </loop>
    </htm>
  </simulation>
</epsn>
```

Listing A.4 – Description en XML de la THG de la simulation LibMultiScale

```
simulation_name = "LibMultiScale";
xml_filename = "./config-epsn/LibMultiScale.xml";

Communicator & com = Parent::dom.getCommunicator();
int GID = Parent::dom.GetGroupID();

std::stringstream sstr;
sstr <<  Parent::dom.myName() << GID;
code_name = sstr.str();

if(com.IsInGroup(lm_my_proc_id, GID)) {
  nb_procs = com.getNBprocsOnGroup(GID);
  my_rank = com.GroupRank(lm_my_proc_id, GID);

  nb_groups = com.getNBGroups();
  num_group = GID;

  if(my_rank == 0) {
    epsn_status = epsn_itfc->initProxyAndNode(simulation_name, xml_filename,
                                              my_rank, nb_procs,
                                              code_name, num_group, nb_groups,
                                              (lm_my_proc_id == 0));
  } else {
    epsn_status = epsn_itfc->initNode(simulation_name, my_rank, nb_procs,
                                      code_name, num_group, nb_groups);
  }
 }
```

Listing A.5 – Initialisation de la plate-forme EPSN dans la simulation LibMultiScale

```
RedSYM::NumberingStyle numbering_style = RedSYM::_fortran_numbering;
RedSYM::CellType element_type = RedSYM::_triangle;

mesh = new RedSYM::Mesh(this->name,
                        my_rank,
                        nb_procs,
                        Dim,
                        element_type,
                        numbering_style);

int mesh_cell_serie = mesh->addVariable("cells", RedSYM::_long,   elt_type);
int mesh_node_serie = mesh->addVariable("nodes", RedSYM::_double, Dim);
int mesh_displacement_serie = mesh->addVariable("displacements",RedSYM::_double, Dim);

mesh->setCellVariable(mesh_cell_serie);

mesh->setNodeVariable(mesh_node_serie);
mesh->setNodeDataVariable(mesh_displacement_serie);

RedSYM::Key region = mesh->addRegion(elements, nodes);
mesh->setRegionTag(region, lm_my_proc_id*100);

mesh->setNumberOfCells(region, elements, elements);
mesh->setNumberOfNodes(region, nodes, nodes);

mesh->wrapCells(region, connectivity);
mesh->wrapNodes(region, coordinates);
mesh->wrap(mesh_displacement_serie, region, displacements);
```

Listing A.6 – Description du maillage à l'aide de la l'API de la bibliothèque RedSYM

```
epsn_itfc->beginHTM();
epsn_itfc->beginLoop("main");

for (; current_step < nb_step ; ++current_step)
  {
    epsn_itfc->beginTask("body");

    stimulator.Stimulate(PRE_POS_VEL_UPDATE);

    epsn_itfc->beginTask("dump");
    dumper.Dump();
    epsn_itfc->endTask("dump");

    epsn_itfc->beginTask("updatepositions");
    dom.UpdatePositions();
    epsn_itfc->endTask("updatepositions");

    stimulator.Stimulate(PRE_FORCE_UPDATE);

    epsn_itfc->beginTask("updateforces");
    dom.UpdateForces();
    epsn_itfc->endTask("updateforces");

    stimulator.Stimulate(PRE_VEL_UPDATE);

    epsn_itfc->beginTask("updatevelocities");
    dom.UpdateVelocities();
    epsn_itfc->endTask("updatevelocities");

    stimulator.Stimulate(PRE_COUPLING_UPDATE);

    epsn_itfc->beginTask("updatecoupling");
    dom.UpdateStuck();
    epsn_itfc->endTask("updatecoupling");

```

```
36      epsn_itfc->endTask("body");
37    }
38
39  epsn_itfc->endLoop("main");
40  epsn_itfc->endHTM();
```

Listing A.7 – Instrumentation de la boucle principale de la simulation LibMultiScale

Annexe B

Bibliographie principale

[1] J. Ahrens, C. Law, W. Schroeder, K. Martin et M. Papka : A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets. Rap. tech. LAUR-001630, Los Alamos National Laboratory, 2000.

[2] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt et J. A. Kohl : The CCA Core Specification in a Distributed Memory SPMD Framework. *Concurrency and Computation : Practice and Experience*, 14(5):323–345, 2002.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher et S. Mock : Kepler : An Extensible System for Design and Execution of Scientific Workflows. *In In SSDBM*, p. 21–23, 2004.

[4] Amira : An Advanced 3D Visualization and Volume Modeling System. http ://www.amiravis.com.

[5] G. Anciaux : *Simulation multi-échelles des solides par une approche couplée dynamique moléculaire/éléments finis. De la modélisation à la simulation haute performance*. Informatique, Université de Bordeaux 1, janvier 2007.

[6] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker et B. Smolinski : Toward a Common Component Architecture for High-Performance Scientific Computing. *In HPDC '99 : Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, p. 115–124. IEEE Computer Society, 1999.

[7] AVS : Advanced Visual Systems Inc. http ://www.avs.com.

[8] P. H. Beckman, P. K. Fasel, W. F. Humphrey et S. M. Mniszewski : Efficient Coupling of Parallel Applications Using PAWS. *In HPDC '98 : Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, p. 215, Washington, DC, USA, 1998. IEEE Computer Society.

[9] T. Beisel, E. Gabriel et M. Resch : An Extension to MPI for Distributed Computing on MPPs. *In Proceedings of the 4th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, p. 75–82. Springer-Verlag, 1997.

[10] D. Bennett et P. A. Farrell : Experiences Instrumenting a Distributed Molecular Dynamics Program. *In Proceedings of PACISE'O6*, p. 45–50, 2006.

[11] D. Bennett, P. A. Farrell et C. Stein : A Chromium Based Viewer for CUMULVS. *In The Proceedings of the 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'06)*, p. 472–477, 2006.

[12] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus et S. Zhou : A Component Architecture for High-Performance Scientific Computing. *Intl. J. High-Perf. Computing Appl.*, 2004. Submitted to ACTS Collection special issue.

[13] F. Bertrand, R. Bramley, A. Sussman, D. E. Bernholdt, J. A. Kohl, J. W. Larson et K. B. Damevski : Data Redistribution and Remote Method Invocation in Parallel Component Architectures. *Parallel and Distributed Processing Symposium, International*, 1:40b, 2005.

[14] The Bespoke Framework Generator (BFG). http ://intranet.cs.man.ac.uk/cnc/projects/bfg.php.

[15] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning et A. R. Porter : Computational Steering in RealityGrid. *In Proceedings of the UK e-Science All Hands Meeting*, 2003.

[16] T. Bulatewicz et J. Cuny : InCouple : Support For Model Coupling. http ://www.incouple.net/.

[17] T. Bulatewicz et J. Cuny : A domain-specific language for model coupling. *In WSC '06 :q Proceedings of the 38th conference on Winter simulation*, p. 1091–1100. Winter Simulation Conference, 2006.

[18] T. Bulatewicz, J. Cuny et M. Warman : The potential coupling interface : metadata for model coupling. *In WSC '04 : Proceedings of the 36th conference on Winter simulation*, p. 183–190. Winter Simulation Conference, 2004.

[19] CACTUS. http ://www.cactuscode.org.

[20] CCSM Research Tools : Community Atmosphere Model (CAM). http ://www.ccsm.ucar.edu/models/atm-cam/.

[21] CCSM : Community Climate System Model. http ://www.ccsm.ucar.edu.

[22] Chromium Homepage. http ://chromium.sourceforge.net/.

[23] Community Land Model. http ://www.cgd.ucar.edu/tss/clm/.

[24] COMSOL : COMSOL - Multiphysics Modeling. http ://www.comsol.fr/.

[25] O. Coulaud, M. Dussere et A. Esnard : Toward a Computational Steering Environment based on CORBA. *In* G. Joubert, W. Nagel, F. Peters et W. Walter, éds : *Parallel Computing : Environments And Tools for Parallel Scientific Computing*, vol. 13 de *Advances in Parallel Computing*, p. 151–158, Dresden, Germany, 2004. Elsevier.

[26] P. Coveney, G. D. Fabritiis, M. Harvey, S. Pickles et A. Porter : On steering coupled models. *In UK e-Science All Hands Meeting 2005*, 2005.

[27] P. V. Coveney, G. D. Riley et R. W. Ford : Hybrid molecular-continuum fluid models : implementation within a general coupling framework. *Philosophical Transactions Series A : Mathematical, Physical and Engineering Sciences*, 363 (1833):1975 – 1985, aug 2005.

[28] CCSM Research Tools : Community Sea Ice Model (CSIM). http ://www.ccsm.ucar.edu/models/ice-csim/.

[29] K. Damevski et S. G. Parker : Parallel Remote Method Invocation and M-by-N Data Redistribution. *In 12th High-Performance Distributed Computing Conference (HPDC)*, 2003.

[30] A. Denis, C. Pérez et T. 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)*, p. 835–844, 2001.

[31] F. Dijkstra et A. J. van der Steen : Integration of two ocean models within Cactus. *Concurr. Comput. : Pract. Exper.*, 18(2):193–202, 2006.

[32] The DL_POLY molecular simulation package.
http ://www.cse.scitech.ac.uk/ccg/software/DL_POLY/.

[33] Molecular Dynamics on Massively Parallel Computers. http ://www.fz-juelich.de/jsc/cv/mdproject/.

[34] G. Edjlali, A. Sussman et J. Saltz : Interoperability of Data Parallel Runtime Libraries with Meta–Chaos. Rap. tech. CS-TR-3633 and UMIACS-TR-96-30, University of Maryland, Department of Computer Science and UMIACS, mai 1996. A condensed version submitted to Supercomputing'96.

[35] G. E.Fagg et J. J. Dongarra : PVMPI : An Integration of the PVM and MPI Systems. Rap. tech., University of Tennessee, 1996.

[36] T. Eickermann, W. Frings, P. Gibbon et et al. : Steering UNICORE applications with VISIT. *Royal Society of London Philosophical Transactions Series A*, 363:1855–1865, août 2005.

[37] Ensight. http ://www.ceintl.com/products/ensight.html.

[38] EPSN Project. http ://www.labri.fr/projet/epsn/index.html.

[39] ESMF : Earth System Modeling Framework. http ://www.esmf.ucar.edu.

[40] 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.

[41] A. Esnard : RedGRID : un environnement pour la redistribution d'objets complexes. *Technique et Science Informatiques (TSI)*, 27(3-4):427–455, 2008.

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

[43] A. Esnard, N. Richart et 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)*, p. 7–14, Torremolinos, Malaga, Spain, October 2006. IEEE Press.

[44] J. M. Favre : Large Data and Distributed Visualization with the Visualization Toolkit (VTK). *EPFL Supercomputing Review*, p. 7–11, january 2004.

[45] FluidBox. http ://www.math.u-bordeaux.fr/ nkonga/FluidBox.htm.

[46] R. W. Ford, G. D. Riley, M. K. Bane, C. W. Armstrong et T. L. Freeman : GCF : a General Coupling Framework. *Concurrency and Computation : Practice & Experience*, 18 (2):163–181, jan 2006.

[47] C. Forum : The Common Component Architecture Forum. http ://www.cca-forum.org/.

[48] D. Foulser : IRIS Explorer : A Framework for Investigation. *SIGGRAPH Comput. Graph.*, 29(2):13–16, 1995.

[49] Cosmological simulations width GADGET. http ://www.mpa-garching.mpg.de/gadget/.

[50] E. Gallopoulos, E. Houstis et J. Rice : Computer as thinker/doer : problem-solving environments for computational science. *Computational Science & Engineering, IEEE*, 1(2):11–23, Summer 1994.

[51] G. A. Geist, I. James, A. Kohl et P. M. Papadopoulos : CUMULVS : Providing fault tolerance, visualization, and steering of parallel applications. *International Journal of High Performance Computing Applications*, 11:224–236, 1997.

[52] M. Girkar et C. D. Polychronopoulos : The hierarchical task graph as a universal intermediate representation. *Int. J. Parallel Program.*, 22(5):519–551, 1994.

[53] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel et J. Shalf : The Cactus Framework and Toolkit : Design and Applications. *In Vector and Parallel Processing - VECPAR '2002, 5th International Conference*. Springer, 2003.

[54] C. Goodrich, A. Sussman, J. Lyon, M. Shay et P. Cassak : The CISM code coupling strategy. *Journal of Atmospheric and Solar-Terrestrial Physics*, 66(15-16):1469 – 1479, 2004. Towards an Integrated Model of the Space Weather System.

[55] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca et A. Lumsdaine : Open MPI : A high-performance, heterogeneous MPI. *In Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

[56] Grid5000 :Home. https ://www.grid5000.fr/.

[57] W. Gropp, E. Lusk et A. Skjellum : *Using MPI (2nd ed.) : Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[58] R. Haber, B. Bliss, D. Jablonowski et C. Jog : A Distributed Environment for Run-Time Visualization and Application Steering in Computational Mechanics. *Computing Systems in Engineering*, p. 501–515, 1992.

[59] C. Hansen et P. Hinker : Massively Parallel Isosurface Extraction. *In* IEEE Computer Society Press, éd. : *Proceedings of Visualization '92*, p. 77–83, 1992.

[60] HDF Group - HDF5. http ://www.hdfgroup.org/HDF5/.

[61] P. S. Heckbert et M. Garland : Surface Simplification using Quadric Error Metrics. *In ACM SIGGRPAH '97*, p. 209–216, 1997.

[62] P. S. Heckbert et M. Garland : Survey of Polygonal Simplification Algorithms. *In SigGraph '97*, 1997.

[63] C. Hill, C. DeLuca, Balaji, M. Suarez et A. Da Silva : The architecture of the Earth System Modeling Framework. *Computing in Science & Engineering*, 6(1):18–28, Jan-Feb 2004.

[64] P. Hinker et C. Hansen : Geometric Optimization. *In* IEEE Computer Society Press, éd. : *Proceedings of Visualization '93*, 1993.

[65] G. Humphreys, M. Houston, Y. Ng, R. Frank, S. Ahern, P. Kirchner et J. Klosowski : Chromium : A Stream Processing Framework for Interactive Graphics on Clusters. SIGGRAPH, San Antonio, Texas, 2002.

[66] IBM : IBM Deep Computing : Deep Computing Visualization (DCV). http ://www-03.ibm.com/systems/deepcomputing/visualization/.

[67] IBM : OpenDX. http ://www.research.ibm.com/dx.

[68] InterComm. http ://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/ic/.

[69] J. Dangarra et al. : MPI : A Message-Passing Interface Standard. University of Tennesse, juin 1995.

[70] R. Jacob, J. Larson et E. Ong : M X N Communication and Parallel Interpolation in Community Climate System Model Version 3 Using the Model Coupling Toolkit. *Int. J. High Perform. Comput. Appl.*, 19(3):293–307, 2005.

[71] R. Jacob, J. Larson, E. Ong, R. Jacob et J. Larson : The model coupling toolkit. *In of Lec. Nt. in Comp. Sci., pg 185-194*, p. 185–194. Springer-Verlag, 2001.

[72] X. Jiao, M. T. Campbell et M. T. Heath : Roccom : an object-oriented, data-centric software integration framework for multiphysics simulations. *In ICS '03 : Proceedings of the 17th annual international conference on Supercomputing*, p. 358–368, New York, NY, USA, 2003. ACM.

[73] W. Joppich et M. Kürschner : MpCCI : a Tool for the Simulation of Coupled Applications. *Concurrency and Computation : Practice and Experience*, 18(2):183–192, 2006.

[74] K. Keahey : PAWS : Collective Interactions and Data Transfers. *In in Proceedings of the High Performance Distributed Computing Conference*, p. 47–54, 2001.

[75] K. Keahey et D. Gannon : PARDIS : A Parallel Approach to CORBA. *In HPDC*, p. 31–39, 1997.

[76] An Extensible System for Scientific Workflows. http ://kepler.ecoinformatics.org.

[77] C. Koebel, D. Loveman, R. Schreiber, G. S. Jr. et M. Zosel : *The High Performance Fortran Handbook*. MIT Press, Cambridge, 1994.

[78] J. A. Kohl et P. M. Papadopoulos : Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS. *In 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998.

[79] S. Krishnan, R. Bramley, D. Gannon, M. Govindaraju, R. Indurkar, A. Slmonski, B. Temko, J. Alameda, R. Alkire, T. Drews et E. Webb : The XCAT Science Portal. *In Proceedings of the 2001 ACM/IEEE conference on Supercomputing (SC2001)*, p. 49–49, Denver, Colorado, November 2001. ACM SIGARCH, ACM Press.

[80] A. N. Laboratory : MPICH2 : High-performance and Widely Portable MPI. http ://www.mcs.anl.gov/research/projects/mpich2/.

[81] Lammps molecular dynamics simulator. http ://lammps.sandia.gov/index.html.

[82] C. Law : A Multithreaded Streaming Pipeline Architecture for Large Structured Meshes. *IEEE Visualization Proceedings*, 1999.

[83] C. C. Law, A. Henderson et J. Ahrens : An Application Architecture for Large Data Visualization : A Case Study. *In PVG '01 : Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, p. 125–128, Piscataway, NJ, USA, 2001. IEEE Press.

[84] J.-Y. Lee et A. Sussman : Efficient Communication Between Parallel Programs with InterComm. Rap. tech. CS-TR-4557 and UMIACS-TR-2004-0, University of Maryland, Department of Computer Science and UMIACS, January 2004.

[85] V. Mann, V. Matossian, R. Muralidhar et M. Parashar : DISCOVER : An Environment for Web-based Interaction and Steering of High-Performance Scientific Applications. *Concurrency and Computation : Practice and Experience*, 13(8-9):737–754, 2001.

[86] R. Marshall, J. Kempf, S. Dyer et C.-C. Yen : Visualization methods and simulation steering for a 3D turbulence model of Lake Erie. *In Proceedings of the 1990 symposium on Interactive 3D graphics*, vol. 24, p. 89–97. ACM Press, 1990.

[87] B. McCormick, T. DeFanti et M. Brown : Visualization in scientific computing. *Computer Graphics*, 21(6), november 1987.

[88] MCT : The Model Coupling Toolkit. http ://www-unix.mcs.anl.gov/mct.

[89] G. Mercier : MPICH-Madeleine : An MPI Implementation for Heterogeneous Clusters of Clusters. http ://runtime.bordeaux.inria.fr/mpi/.

[90] G. Mercier, F. Trahay, D. Buntinas et E. Brunet : NewMadeleine : An Efficient Support for High-Performance Networks in MPICH2. *In Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, Rome, Italy, mai 2009. IEEE Computer Society Press. To appear.

[91] Message-Passing Interface Forum : MPI-2.0 : Extensions to the Message-Passing Interface, june 1997.

[92] Microsoft : COM : Component Object Model Technologies. http ://www.microsoft.com/com.

[93] S. Microsystems : Remote Procedure Call specification version 2 (RPC). http ://www.faqs.org/rfcs/rfc1057.html, june 1988.

[94] K. Moreland et D. Thompson : From cluster to wall with VTK. *IEEE symposium on Parallel and Large-Data Visualization and Graphics*, p. 25–31, 2003.

[95] K. Moreland, B. Wylie et C. Pavlakos : Sort-Last Parallel Rendering for Viewing Extremely Large Data Sets on Tile Displays. *In PVG '01 : Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, p. 85–92. IEEE Press, 2001.

[96] MpCCI : Mesh-based parallel code coupling Interface. http ://www.mpcci.org.

[97] J. D. Mulder : *Computational steering with parametrized geometric objects*. Thèse de doctorat, University of Amsterdam, june 1998.

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

[99] R. Muralidhar, S. Kaur et M. Parashar : An Architecture for Web-Based Interaction and Steering of Adaptive Parallel Distributed Applications. *Lecture Notes in Computer Science*, 1900, 2001.

[100] R. Muralidhar et M. Parashar : An Interactive Object Infrastructure for Computational Steering of Distributed Simulations. Proceedings of the ninth IEEE International Symposium on High Performance Distributed Computing (HPDC), 2000.

[101] R. D. Muralidhar : *A Distributed Object Framework for the Interactive Steering of High-Performance Applications*. Thèse de doctorat, Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey, 2000.

[102] . http ://nossi.gforge.inria.fr/.

[103] OMG : Object Management Group. http ://www.omg.org.

[104] OMG : Data Parallel CORBA Specification v1.0. OMG : Object Management Group, November 2006.

[105] OMG : CORBA 3.0.3, The Common Object Request Broquer Architecture : Core Specification 12-03-04. http ://www.omg.org/docs/formal/04-03-12.pdf, december 2001.

[106] OMG : CORBA Components, v3.0 (full specification), Specification 02-06-65. http ://www.omg.org/technology/documents/formal/components.htm, june 2002.

[107] OMG : Object Management Group : Common Object Request Broker Architecture Specification. http ://www.corba.org.

[108] OpenGL – The Industry's Foundation for High Performance Graphics. http ://www.opengl.org/.

[109] Open MPI : Open Source High Performance Computing. http ://www.open-mpi.org/.

[110] ParaView. http ://www.paraview.org.

[111] S. Parker : *The SCIRun Problem Solving Environment and Computational Steering Software System*. Thèse de doctorat, University of Utah, august 1999.

[112] S. Parker, C. Johnson et D. Beazley : Computational steering. Software systems and strategies. *Computational Science & Engineering, IEEE*, 4(4):50–59, Oct-Dec 1997.

[113] S. Parker, M. Miller, C. Hansen et C. Johnson : Computational Steering and the SCIRun Integrated Problem Solving Environment. *In* H. Hagen, G. Nielson et F. Post, éds : *Proceedings of Dagstuhl 1997 Workshop on Scientific Visualization*, p. 257–266, 2000. Invited and peer reviewed.

[114] C. Pérez, T. Priol et A. Ribes : PaCO++ : A Parallel Object Model for High-Performance Distributed Systems. *In Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, Big Island, Hawaii, USA, January 2004. IEEE Computer Society Press.

[115] A. Piacentini et T. P. Group : PALM : A Dynamic Parallel Coupler. *In* S. B. . Heidelberg, éd. : *High Performance Computing for Computational Science – VECPAR 2002*, vol. 2565/2003 de *Lecture Notes in Computer Science*, p. 355–367, Porto, Potugal, June 2002.

[116] S. M. Pickles, R. Haines, R. L. Pinning et et al. : A practical toolkit for computational steering. *Royal Society of London Philosophical Transactions Series A*, 363:1843–1853, août 2005.

[117] CCSM Research Tools : CCSM3.0 POP Documentation. http ://www.ccsm.ucar.edu/models/ccsm3.0/pop/.

[118] Prism - Trac. https ://prismtrac.cerfacs.fr/.

[119] Qt - A cross-platform application and UI framework. http ://www.qtsoftware.com/.

[120] M. Ranganathan, A. Acharya, G. Edjlali, A. Sussman et J. Saltz : Runtime Coupling of Data-Parallel Programs. *In ICS '96 : Proceedings of the 10th international conference on Supercomputing*, p. 229–236, New York, NY, USA, 1996. ACM Press.

[121] S. Rathmayer et M. Lenke : A Tool for On-line Visualization and Interactive Steering of Parallel HPC Applications. *In Proceedings of the 11th IPPS'97*, p. 181–186, 1997.

[122] EPSN Project. http ://www.labri.fr/projet/epsn/index.html.

[123] Site web du GIP RENATER. https ://www.renater.fr/.

[124] M. Riedel, W. Frings, S. Dominiczak, T. Eickermann, D. Mallmann, P. Gibbon et T. Dussel : VISIT/GS : Higher Level Grid Services for Scientific Collaborative Online

Visualization and Steering in UNICORE Grids. *In ISPDC ’07 : Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, p. 12, Washington, DC, USA, 2007. IEEE Computer Society.

[125] Roccom Users and Developpers Page. http ://www.cse.uiuc.edu/ jiao/Roccom/.

[126] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee et H. Casanova : *Overview of GridRPC : A Remote Procedure Call API for Grid Computing*, vol. 2536/2002, chap. Overview of GridRPC : A Remote Procedure Call API for Grid Computing, p. 274–278. Springer Berlin / Heidelberg, 2002.

[127] Siesta Project : Siesta - home. http ://www.icmab.es/siesta/index.php.

[128] C. Stein, D. Bennett, P. A. Farrell et A. Ruttan : A Steering and Visualization Toolkit for Distributed Applications. *In The Proceedings of the 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’06)*, p. 451–457, 2006.

[129] A. Sussman : Building complex coupled physical simulations on the grid with InterComm. *Eng. with Comput.*, 22(3):311–323, 2006.

[130] C. Szyperski : *Component Software : Beyond Object-Oriented Programming.* ACM Press/Addison-Wesley Publishing Co., 1998.

[131] TAU - Tuning and Analysis Utilities. http ://www.cs.uoregon.edu/research/tau/home.php.

[132] Tcl Developper Site. http ://www.tcl.tk/.

[133] TOP500 Supercomputing Sites. http ://www.top500.org.

[134] Tulip Software home page. http ://tulip.labri.fr/.

[135] S. Valcke, E. Guilyardi et C. Larsson : PRISM and ENES : a European approach to Earth system modelling. *Concurr. Comput. : Pract. Exper.*, 18(2):247–262, 2006.

[136] J. Vetter et K. Schwan : Progress : A Toolkit for Interactive Program Steering. *In Proceedings of the 1995 International Conference on Parallel Processing*, p. 139–142, 1995.

[137] J. Vetter et K. Schwan : Models for Computational Steering. Rap. tech. GIT-CC-95-39, Georgia Institute of Technology, 1996.

[138] J. Vetter et K. Schwan : High Performance Computational Steering of Physical Simulations. *In Proceedings of the 11th IPPS’97*, p. 128–132, 1997.

[139] VISIT - a Visualization Interface Toolkit. http ://www.fz-juelich.de/jsc/visit/.

[140] VisIt - Visualisation Tool. https ://wci.llnl.gov/codes/visit/home.html.

[141] VTK : The Visualization Toolkit. http ://www.vtk.org.

[142] D. Weinstein, S. Parker, J. Simpson, K. Zimmerman et G. Jones : Visualization in the SCIRun Problem-Solving Environment. *In* C. Hansen et C. Johnson, éds : *The Visualization Handbook*, p. 615–632. Elsevier, 2005.

[143] A. Wollrath, R. Riggs et J. Waldo : A Distributed Object Model for the Java System. *In 2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, p. 219–232. USENIX Association, 1996.

[144] B. Wylie, C. Pavlakos, V. Lewis et K. Moreland : Scalable Rendering on PC Clusters. *IEEE Comput. Graph. Appl.*, 21(4):62–70, 2001.

[145] K. Zhang, K. Damevski et S. G. Parker : SCIRun2 : A CCA framework for high performance computing. *In In Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS*, p. 72–79. IEEE Press, 2004.

Annexe C

Liste des publications

[1] A. Esnard, N. Richart et 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.

[2] N. Richart, A. Esnard et O. Coulaud : A Steering Environment for Legacy Coupled SPMD Simulations. En cours de rédaction.

[3] N. Richart, A. Esnard et O. Coulaud : Toward a Computational Steering Environment for Legacy Coupled Simulations. *In Proceedings of 6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, pages 319–326, Hagenberg, Austria, July 2007. IEEE Press.

Glossaire

Symbols

$\mathcal{D}_P$: *Ensemble des dates de points* 46

$\mathcal{D}_T$: *Ensemble des dates de tâches* 45

$\mathcal{D}_T(m)$: *Ensemble des dates de tâches restreinte au masque de date m* 47

$\mathcal{R}_D(\Pi)$: *Ensemble des révisions de la donnée D dans le THP Π* 47

F

FTHL : *Forêt de Tâches Hiérarchiques Locale* 54

H

HPC : *High-Performance Computing* 2

M

M-SPMD : *Simulation de type Multiple-SPMD* 9

MHT : *Modèle Hiérarchique en Tâches* 43

MHTd : *Modèle Hiérarchique en Tâches pour la représentation de simulations distribuées* 53

MHTp : *Modèle Hiérarchique en Tâches pour la représentation de simulations parallèles* 53

T

THG : *Tâche Hiérarchique Globale* 53

THL : *Tâche Hiérarchique Locale* 54

THP : *Tâche Hiérarchique Principale* 44