Conception par objets d'applications parallèles réparties

Jean-Marc Jézéquel

IRISA/CNRS
Campus de Beaulieu
F-35042 Rennes Cedex, FRANCE
E-mail: jezequel@irisa.fr ;
http://www.irisa.fr/pampa/PROF/jmj.html
Tel: +33 299847192 Fax: +33 299847171

Résumé

La parallélisation d'applications de grande taille nécessite de plus en plus souvent la maîtrise de différentes formes d'irrégularité, tant au niveau de l'application que du ou des supports d'exécution. Nous montrons comment une approche résolument orientée objet et fondée sur l'utilisation de patrons de conceptions spécifiques permet d'exprimer des traitements à un niveau où l'irrégularité des structures sous-jacentes est abstraite; ce qui permet de les exécuter optimalement tant sur des architectures centralisées que parallèles à mémoire (virtuellement) partagée ou distribuée, ou même mixtes (méta-computing). Cette présentation s'appuyera sur l'exemple de l'environnement EPEE, qui offre un cadre de conception, basé sur le langage Eiffel, pour développer des composants logiciels réutilisables pour applications traitant de grandes quantités d'événements, de données ou de calculs.

1 Introduction

De nombreux programmeurs sont intéressés par la puissance de calcul offerte par les architectures multi-processeurs mais restent réticents à l'idée de transférer leurs applications vers ces dernières. En effet, les outils logiciels associés à ces machines manquent cruellement de maturité et les méthodes et environnements de programmation adaptés aux machines séquentielles s'avèrent peu appropriés à la gestion du parallélisme. Pour bien exploiter

ces architectures parallèles, le programmeur doit ainsi posséder une connaissance approfondie de l'algorithmique parallèle et des spécificités de l'architecture utilisée.

L'objectif de cette présentation est de montrer comment une approche résolument orientée objet et fondée sur l'utilisation de patrons de conceptions spécifiques permet d'exprimer des traitements à un niveau où l'irrégularité des structures sous-jacentes est abstraite; ce qui permet de les exécuter optimalement tant sur des architectures centralisées que parallèles à mémoire (virtuellement) partagée ou distribuée, ou même mixtes (méta-computing). Cette présentation s'appuyera sur l'exemple de l'environnement EPEE, qui offre un cadre de conception, basé sur le langage Eiffel, pour développer des composants logiciels réutilisables pour applications traitant de grandes quantités d'événements, de données ou de calculs. On montrera comment appliquer ces idées sur une étude de cs simple : le problème des N corps.

2 Approche par objets et parallélisme et répartition

2.1 Approches classiques

Si la construction de systèmes complexes à l'aide des technologies objets commence à être assez bien maîtrisée, en revanche le bât blesse encore pour tout ce qui touche au parallélisme et à la distribution. En effet, comme l'avait écrit dès 1993 B. Meyer dans [18],

> *To judge by the looks of the two parties, the marriage between concurrent computation and object-oriented programming appears an easy enough affair to arrange. This appearance is deceptive: the problem is a hard one.*

En effet, dans les langages à objets, un objet est une unité de structuration de programme encapsulant données et méthodes (procédures et fonctions) travaillant sur ces données. En utilisant la terminologie Smalltalk, on dit que des objets communiquent par envoi de messages. On voit donc apparaître un certain nombre de points communs avec la notion de processus. La tentation est donc forte d'intégrer ces deux notions dans celle d'objet actif. POOL-T [1] est certainement un des premiers exemples significatifs illustrant cette approche : une fois créé, un objet peut continuer à être actif après avoir rendu la main à son créateur. La communication entre objets est ici de type Remote Procedure Call (appel de procédure à distance), et

le programmeur contrôle alors explicitement le parallélisme et l'accès aux données.

Un autre moyen pour introduire du parallélisme dans les langages à objets consiste à rendre asynchrones les appels aux méthodes des objets : un objet appelant la méthode d'un autre objet peut continuer son activité en parallèle avec l'objet exécutant la méthode appelée. Ceci est implanté par exemple dans ABCL/1 [21], ELLIE [2], Eiffel// [4], ou encore ConcurrentSmalltalk [20].

Ces deux idées d'objet actif et d'appels de procédure asynchrones sont utilisées pour créer des LAO parallèles soit en intégrant ce parallélisme dès la phase de conception d'un nouveau langage de programmation concurrent (ELLIE, POOL-T, ABCL/1, ou plus récemment Java), ou plus simplement en étendant des langages séquentiels déjà existants pour leur permettre de gérer le parallélisme, comme dans DistributedSmalltalk [3] qui étend Smalltalk ou COOL [5] qui étend C++, ou encore dans CORBA, qui fourni les mécanismes par lesquels des objets (implantés dans différents langages) peuvent émettre des requêtes et recevoir des réponses de façon transparente.

2.2 Problématique

Le modèle de parallélisme sous-jacent à ces langages (fondamentalement les processus séquentiels communicants de Hoare [9]) pose de nombreux problèmes liés à sa nature, à la fois très primitif et très (trop) général.

Tout d'abord, il se concilie difficilement avec le mécanisme d'héritage inhérent aux langages à objets, en particulier en ce qui concerne les contraintes de synchronisation. Ce problème, connu sous le nom d'*anomalie d'héritage* [16] ne semble pas pouvoir être facilement contournable sans restreindre l'expressivité du modèle [17]. En effet ce problème est étroitement lié à la nécessité d'imposer des contraintes sur l'acceptation d'un message par un objet (exemple du tampon borné, dans [17]). Dans la plupart des langages à objets prenant en compte la concurrence, ceci est réalisé par du code de synchronisation contrôlant l'acceptation d'un message en fonction de l'état de l'objet récepteur. Ces codes de synchronisation sont difficiles à hériter et demandent généralement une redéfinition *in extenso*. L'anomalie d'héritage [16] est un problème causé par la présence de ce code de synchronisation, qu'il faut donc chercher à éliminer. Une approche possible consiste en l'intégration étroite d'une sémantique particulière du parallélisme dans un langage. C'est l'exemple de Parallel Eiffel [18], où il n'y a plus de distinction entre objets "actifs" et "passifs", mais seulement la possibilité qu'un objet soit "séparé" (*separate*), c'est à dire traité par un autre processeur (réel

ou virtuel). Cependant Parallel Eiffel s'appuie sur un mécanisme de multi rendez-vous réparti, qui est bien entendu suffisamment puissant pour permettre d'apporter des solutions simples, concises et élégantes aux problèmes classiques de concurrence (e.g. le problème dit *des philosophes*), mais qui limite drastiquement le domaine d'intérêt de l'approche.

D'autre part, comme les communications (et les synchronisations) entre processus sont entièrement sous la responsabilité du programmeur, celui-ci est confronté de plein fouet à la complexité des systèmes répartis : interblocages, absence d'état global observable rendant difficile le débogage réparti ou même simplement la détection de la terminaison, etc.

2.3 Entre langages et systèmes, la notion de "framework"

S'il n'est donc pas raisonnable, d'un point de vue génie logiciel, de programmer "à la main" (c'est à dire à un niveau sémantique aussi primitif que celui des processus communicants) des systèmes "critiques" de grande taille, subissant ou utilisant la répartition, il paraît tout aussi difficile, et surtout peu rentable, de rechercher un langage universel prenant en compte tous les aspects possibles de leur programmation. Ne serait-ce qu'en termes de sémantique de la communication entre processus (rendez-vous, files, RPC, hypothèses de fiabilité, d'ordonnancement, d'isochronisme, etc.) quelque soit le choix effectué, il serait forcément fermé (puisque intégré dans un langage), donc trop limitatif pour certaines applications, et par essence inapproprié pour des systèmes parallèles répartis.

Il parait en revanche beaucoup plus prometteur de définir (à l'aide de la technologie objet, qui est particulièrement bien adaptée pour cela) des *modèles* spécifiques à des domaines d'applications particuliers, comme par exemple le modèle d'exécution SPMD associé à la distribution de données pour le calcul scientifique intensif; et de fournir des cadres de conception, de réalisation et de validation adaptés à ces modèles : c'est la notion de "framework".

Un *framework* fournit un ensemble intégré (tout en restant ouvert et extensible) de fonctionnalités spécifiques à un domaine. Il consiste en une collection de classes liées entre elles par de multiples schémas (*patterns*) de collaboration statiques et dynamiques [6]. Il fournit un modèle d'interaction entre les différents objets instances des classes définies (ou seulement spécifiées pour les classes abstraites) dans le framework. Celui-ci présente en général une inversion du contrôle à l'exécution : alors qu'une application utilisant une bibliothèque s'appuie sur celle ci, dans le cas d'une application utilisant un framework, c'est le framework qui effectue l'essentiel du

travail et appelle “de temps en temps” un composant spécifique réalisé par l'implanteur de l'application. Un framework peut donc être vu comme une application semi-complète. Des applications complètes sont développées en héritant et en instantiant des composants paramétrés de frameworks. Il suffit donc en quelque sorte d'enficher dans un framework les composants spécifiques de son application pour obtenir une application complète.

Nous explorons depuis quelques années certains de ces problèmes, avec des domaines d'application variés, avec en particulier le calcul scientifique intensif [8, 11, 13, 15]. L'objectif était d'étudier, concevoir et valider des modèles et des méthodes de construction de logiciels pour architectures réparties par composition de composants logiciels dans un contexte de programmation par objets [10].

3 Application à la mise en oeuvre d'applications irrégulières

3.1 Le cadre général de EPEE

Notre approche s'appuie sur le principe de l'encapsulation des aspects liés au parallélisme dans des classes d'un langage à objets séquentiel. Pour cela, nous avons développé l'environnement EPEE (Environnement Parallèle d'Exécution de Eiffel), adoptant un modèle d'exécution SPMD qui permet de décomposer naturellement une application en un entrelacement de phases séquentielles et de phases parallèles. Ces dernières étaient initialement obtenues par réutilisation de composants logiciels encapsulant un parallélisme de données.

La solution que nous proposons, et que nous continuons d'expérimenter sur des problèmes concrets, dans l'environnement EPEE s'appuie sur des travaux à trois niveaux :

Exécutif [14] : L'introduction d'objets partagés dans un langage muni d'un ramasse miette au sein de son exécutif nous a demandé de modifier son comportement afin de disposer du ramasse miettes pour les objets partagés de la même façon que pour les objets alloués dans l'espace d'adressage local de chaque processus.

Boite à outils [19] : La migration du code séquentiel vers du code réparti se fait à l'aide de composants réutilisables intégrant la notion de distribution de données et de contrôle, tout en la masquant par une interface identique à leurs versions séquentielles.

Patrons de conception [15, 12] : Les patrons de conception, ou *design patterns*, introduits dans [6], sont des modèles de solutions permettant de résoudre des problèmes de conception particuliers, notamment dans un contexte de conception par objets de logiciels. Les *design patterns* permettent l'identification et la réutilisation de micro-architectures logicielles, qui sont décrites tant du point de vue statique que dynamique sous la forme d'un ensemble de classes et de leurs relations structurelles et contextuelles (i.e. leurs collaborations). Dans le cadre du parallélisme de données, auquel nous nous attachons ici, nous avons proposé le patron de conception des OPÉRATEURS permettant de construire des bibliothèques de composants qui soient à la fois réutilisables, extensibles et permettant une exécution parallèle (voir ci-dessous).

À travers cette maquette EPEE, nous avons montré l'intérêt de notre approche pour faciliter la programmation d'architectures parallèles réparties : elle permet de présenter un modèle de programmation séquentiel au programmeur d'application, qui ne voit alors la machine parallèle que comme un processeur de calcul plus puissant. Ce modèle s'adapte bien à l'expression d'un parallélisme massif pourvu que les problèmes à résoudre soient de taille assez grande. Nous utilisons maintenant EPEE surtout comme une plate-forme de réflexion et d'intégration pour mener nos recherches sur la notion de modèles de conception s'abstrayant des particularités de l'architecture répartie sous-jacente.

3.2 Les opérateurs parallèles

L'adoption de technologies objets dans la communauté de la programmation parallèle doit permettre à cette dernière de bénéficier des qualités de la conception orientée objet, en particulier la réutilisabilité et la facilité d'extension. Cependant, une approche traditionnelle de la conception des librairies parallèles ne permet pas de satisfaire pleinement ces objectifs. En effet, un principe fréquemment utilisé en conception orientée objet est celui de l'encapsulation qui préconise que des méthodes manipulant des données doivent être encapsulées avec elles dans des classes. Par exemple, lors de la conception d'une classe MATRICE, il est couramment admis qu'une opération comme le produit de matrices doit être définie dans l'interface de la classe MATRICE.

Utilisant ce principe, il est par exemple possible, en utilisant le polymorphisme entre différentes implantations (matrices denses ou creuses, ...) de la classe MATRICE et la liaison dynamique de la méthode *produit()*, réa-

lisant le produit de matrices, de créer des bibliothèques complètes et optimisées, présentant des méthodes adaptées à chaque type d'implantation. Ainsi, la librairie d'algèbre linéaire répartie Paladin [7], développée dans le cadre de l'expérimentation du projet EPEE, a été conçue selon ce principe. Cependant, l'utilisation de cette approche pose des problèmes d'extensibilité (cf. [15]). Ces problèmes vont se poser en particulier lorsque l'utilisateur d'une telle bibliothèque veut ajouter une nouvelle implantation concrète, adaptée à ses besoins, de la classe MATRICE. En effet seules les méthodes génériques seront utilisables lorsque cette nouvelle classe sera utilisée comme argument des méthodes des classes déjà définies.

Pour résoudre ce problème d'extensibilité, il importe de séparer de la classe qui sert de "conteneur" toutes les opérations qui ne dépendent pas de l'implantation de de cette dernière. Ainsi, une méthode *produit* n'a aucun besoin de connaître la façon dont est implantée une matrice du moment qu'elle dispose d'un moyen de lire ses éléments. Les seules opérations qu'une classe conteneur devrait en fait fournir sont celles relatives à la gestion de la structure de données utilisée pour stocker ses éléments. La solution proposée dans [15] est d'opérer une séparation entre les classes liées au domaine de l'application et celles liées au domaine de l'implantation. Pour cela, il importe que les opérations extrinsèques appliquées aux structures de données soient *réifiées*, c'est à dire définies comme des classes propres et non plus comme des méthodes des agrégats manipulés. L'utilisation du *design pattern* de l'opérateur permet de réaliser cette séparation entre les deux domaines cités précédemment.

Ainsi, le *design pattern* de l'opérateur fait intervenir quatre entités de base, deux dépendant du domaine de l'application : les OPÉRATEURS et les ÉLÉMENTS; et deux autres du domaine de l'implantation: les CONTENEURS et les ITÉRATEURS. Les rôles de ces quatre abstractions sont les suivants :

Opérateur : Un OPÉRATEUR représente l'opération régulière appliquée à un agrégats d'ÉLÉMENTS.

Élément : Les ÉLÉMENTS sont les abstractions manipulées par l'application. Ils sont la cible des OPÉRATEURS.

Conteneur : Les CONTENEURS sont chargés de stocker, de modifier et de fournir les ÉLÉMENTS stockés dans une structure de données particulière.

Itérateur : Un ITÉRATEUR permet aux OPÉRATEURS de parcourir les ÉLÉMENTS stockés dans les CONTENEURS. Le rôle de l'ITÉRATEUR est de

masquer la structure concrète des agrégats ainsi que le domaine d'itération. Certains ITÉRATEURs peuvent également être capable de générer eux-mêmes les éléments qu'ils fournissent sans être attachés à un conteneur (e.g. générateur pseudo aléatoire).

Ainsi, un OPÉRATEUR accède aux éléments stockés dans un CONTENEUR par l'intermédiaire d'un ITÉRATEUR. Il est possible pour un OPÉRATEUR, en utilisant le polymorphisme de son ITÉRATEUR, de parcourir un CONTENEUR de différentes façons et d'accéder à ses ÉLÉMENTS indépendemment de la structure interne du CONTENEUR.

Du fait du découplage opéré entre entre les abstractions du domaine de l'application et celles du domaines de l'implantation, la distribution des données va se faire de façon transparente vis-à-vis de l'utilisateur. Ainsi, les applications ne vont pas dépendre d'une architecture particulière et vont être facilement extensibles ou modifiables, toutes les fonctions de gestion du parallélisme (synchronisations, etc.) étant intégrées dans les méthodes d'accès aux données réparties ou partagées.

La migration vers le parallélisme de données d'applications séquentielles utilisant le *design pattern* de l'opérateur va donc se faire simplement en remplaçant les composants séquentiels par des composants polymorphiques incluant la gestion du parallélisme. Ainsi, les opérateurs utilisés sont changés en opérateurs parallèles héritant à la fois des opérateurs concrets et d'un opérateur parallèle générique. De même, le PROVIDER est transformé en PARALLEL PROVIDER adapté à la distribution des données utilisée.

L'utilisation du schéma de conception de l'opérateur dans le cadre d'EPEE permet donc de paralléliser des applications existantes de façon simple et efficace, sans l'utilisation de compilateur spécialisé ou de macro-instructions mais uniquement en tirant en tirant parti des propriétés des langages à objets telles que l'héritage multiple, le polymorphisme ou la liaison dynamique.

3.3 Les conteneurs hiérarchiques et le Méta-Computing

En collaboration avec l'Université de Tokyo (Naohito Sato et les Pr. Yonezawa et Matsuoka), nous avons proposé de nouvelles abstractions permettant d'assurer un découplage entre la distribution des données et le parallélisme [13]. Partant de l'observation qu'un calcul basé sur un modèle d'exécution SPMD peut être considéré comme l'application ordonnée d'une fonction (ou opérateur) sur une collection d'éléments, l'idée est d'abstraire les partitions de données et l'accès à ces données par l'utilisation de struc-

tures de données hiérarchiques, *hierarchical containers*, et d'itérateurs composables, *composable traversers*.

L'introduction des collections hiérarchiques doit permettre d'étendre de façon orthogonale le parallélisme et la distribution afin de pouvoir utiliser facilement et efficacement des schémas de distribution et de parallélisme complexes sans modification du code du client. Les collections hiérarchiques possèdent des relations d'ordre partiel entre les objets situés à un même niveau hiérarchique et des relations d'appartenance entre des collections adjacentes dans la hiérarchie. Les calculs sur des collections hiérarchiques sont effectués à l'aide d'itérateurs qui parcourent les éléments en respectant les relations d'ordre ainsi définies, les objets n'étant pas ordonnés peuvent être traités en parallèle.

Formellement, une collection hiérarchique est définie par l'application sur les éléments d'une partition, notée Π, d'une collection E d'un constructeur C qui groupe ces éléments pour créer un nouveau degré (plus élevé) dans la hiérarchie. Notons $\widehat{C}$, le constructeur composé qui applique C à chaque partition, définie par Π, de E ; une collection hiérarchique peut alors être définie par des applications successives de $\widehat{C}$ à partir d'une collection atomique E^0:

$$E^0 \rightarrow \widehat{C_1} E^1 \rightarrow \widehat{C_2} \cdots$$

De façon similaire, on peut décomposer les itérateurs parcourant les structures de données de l'application à l'aide d'itérateurs de base. A partir de la définition d'une librairie d'itérateurs simples, il devient alors possible en les composant de parcourir de diverses manières les différentes couches des collections hiérarchiques tout en satisfaisant les contraintes de distribution et en maximisant le parallélisme.

Ainsi, selon ces principes, un calcul basé sur l'application d'un opérateur sur une collection d'éléments se fait en décomposant de façon hiérarchique cette dernière afin de séparer parallélisme et distribution et en effectuant une composition d'itérateurs élémentaires pour pouvoir s'adapter à cette configuration particulière.

Les abstractions présentées dans cette section ont été implantées dans le cadre d'EPEE en étendant les notions de *conteneurs* et d' *itérateurs* avec les composants suivants:

Composable Traverser Cette classe permet d'implanter les principes présentés dans le paragraphe précédent.

Partially Ordered Provider Cette extension des *itérateurs* bénéficie des propriétés de composition héritées de la classe *Composable Traverser*.

Hierarchical Container Ce type de conteneur permet de réaliser les translations d'index entre les différents niveaux de la hiérarchie des collections.

Distributed_Container Ce type de conteneur permet d'encapsuler les les détails du modèle de mémoire (mécanisme de gestion du parallélisme et de la distribution, synchronisations, etc.).

L'exécution parallèle du programme est gérée par par l'abstraction *PO_Provider* qui s'occupe des dépendances de données imposées par l'ordre partiel défini sur les éléments à traiter en utilisant les possibilités de synchronisation et de communication offertes par les *Distributed Containers*. Du point de vue de l'utilisateur, l'exécution parallèle du programme est cachée grâce à l'utilisation de composants encapsulant le parallélisme. Le programmeur n'a qu'à manipuler des abstractions de parallélisme. La conception d'un programme parallèle s'effectue alors en trois étapes:

1. définition des éléments et des opérateurs agissant sur ces éléments
2. spécification de la distribution des données à l'aide de collections hiérarchiques
3. compositions d'itérateurs afin de prendre en compte à la fois les dépendances de données et leur distribution physique

Cette méthode permet ainsi au programmeur de définir facilement des schémas relativement complexes de distribution. Ce dernier n'a en effet qu'à choisir les collections et les itérateurs de base définis dans l'extension d'EPEE et de les composer afin de les adapter à son problème. Une application envisagée des collections hiérarchiques de données est le méta-computing, c'est à dire l'utilisation combinée de ressources de calculs hétérogènes (réseaux de stations de travail, calculateurs parallèles, etc.) couplés avec des réseaux à très haut débit. Une première étape concerne la programmation d'architectures à plusieurs niveaux mémoires. Un exemple d'une telle architecture est le *Power Challenge Array* de Silicon Graphics, constitué de l'interconnexion à travers un commutateur HiPPI de système multi-processeurs à mémoire partagée *Power Challenge*. On dispose donc de deux niveaux de mémoire :

- répartie entre les nœuds constitués du système Power Challenge
- et partagée à l'intérieur d'un nœud.

Un autre exemple de ce type d'architecture est la connexion à l'aide d'un réseau Myrinet de serveurs Sparc (ou PC) multi-processeurs. Le réseau Myrinet est un réseau à haut débit (1.28 Gigabits/s), particulièrement bien adapté à l'interconnexion de machines parallèles.

Les collections hiérarchiques ainsi que les itérateurs qui leur sont associés présentent l'avantage de pouvoir refléter cette hiérarchie mémoire et sont donc bien adaptés à la programmation de telles architectures, comme le montrent différentes études de cas que nous avons menées.

4 Exemple d'utilisation : le problème des N corps

Nous présentons dans cette section et la suivante un exemple d'application des idées développées dans ce rapport à travers l'étude de la parallélisation du problème des n-corps. Nous considérons dans cet exemple un ensemble de masses situées dans univers tridimensionnel et qui interagissent entre elles. Chaque masse possède une position ainsi qu'une vitesse et une accélération.

Le but du problème est alors de calculer l'évolution dans le temps de cet ensemble de masses. La simulation est divisée en intervalles de temps. Le calcul de l'évolution pour chaque intervalle se fait en deux étapes. La contribution de chaque masse sur chacun des autres corps est tout d'abord calculée. Ensuite, la position et la vitesse de chaque corps est mise à jour, en tenant compte de toutes les interactions.

4.1 Modélisation du problème

L'univers est défini comme une abstraction capable de stocker des corps ainsi que de fournir des accesseurs de base permettant de les accéder en lecture et écriture. Ces spécifications sont déclarées comme différées dans la classes abstraite UNIVERSE et sont implémentées par la collection instanciable utilisée pour le stockage des corps. La classe abstraite UNIVERSE encapsule également des opérateurs agissant sur tout l'univers, comme celui permettant d'initialiser l'universe de manière aléatoire. Comme le préconise le patron de conception de l'opérateur, ces opérateurs sont inclus sous la forme de *Factory Methods* (voir [6]), afin de pouvoir être rédéfinis facilement.

Les univers instantiables sont alors construits en héritant à la fois de cet univers ainsi que d'une structure de données locale, distribué ou partagée permettant de stocker les corps.

4.2 La mise à jour des corps

La classe UPDATOR est responsable de la mise à jour des paramètres des corps. Cette mise à jour se fait en parcourant l'ensemble des corps et en invoquant la méthode *update* sur chacun d'entre eux. La classe UPDATOR hérite donc de l'opérateur FORALL, la mise à jour constituant la méthode *forall_operation*.

4.3 Le calcul des contributions

Le calcul des interactions entre les différents corps se fait en deux étapes. Tout d'abord, nous calculons la contribution d'une simple masse sur l'ensemble des autres corps. Ce premier calcul est effectué par l'opérateur BODY_CONTRIBUTOR. Ce dernier est ainsi défini comme un opeŕateur de type FORALL qui ajoute la contribution de la masse fournie à tous les corps fourni par son itérateur ; cette opération étant effectuée grâce à la méthode *add_the_contribution_of* de la classe BODY.

La deuxième étape consiste à calculer la contribution d'un ensemble de masses sur un ensemble de corps. Cette opération est effectuée par un deuxième opérateur : UNIV_CONTRIBUTOR, héritant également de FORALL, et qui calcule la contribution de chaque masse fournie par son itérateur sur chaque corps d'un univers donné. Pour chaque masse, l'opérateur UNIV_CONTRIBUTOR utilise donc un BODY_CONTRIBUTOR dans sa méthode *Forall_operation*, comme le montre la figure 1

4.4 Simulation

Tous les composants nécessaires à la simulation ont maintenant été décrits, il ne reste plus qu'a les assembler et à les initialiser, ce qui est le rôle de la classe SIMULATOR. Pour utiliser les opérateurs de la simulation, nous avons besoin de deux itérateurs. Le premier itérateur fournit l'ensemble des masses contenues dans l'univers et est utilisé par la classe UNIV_CONTRIBUTOR. Le second fournit les corps de l'univers et sert aux classes BODY_CONTRIBUTOR et UPDATOR. Comme la classe BODY hérite de MASS, nous pouvons grâce au polymorphisme initialiser ces deux itérateurs de la même manière, en utilisant l'accesseur *items* de l'univers.

5 Parallélisation de la simulation

Nous allons maintenant décrire comment il est possible, en utilisant les abstractions de parallélismes décrites dans le chapitre précédent, de paral-

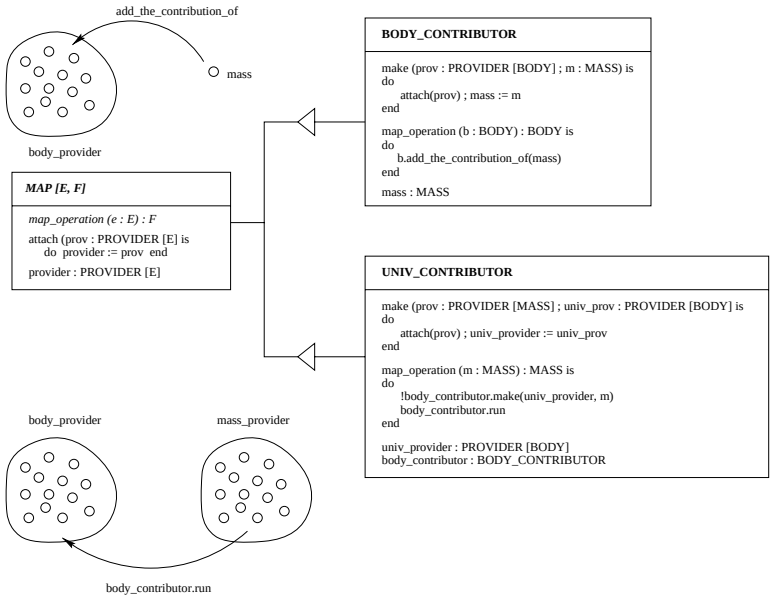

FIG. 1 – *Les opérateurs utilisés pour le calcul des contributions*

léliser la simulation des n-corps sans avoir à réécrire les collections et les opérateurs séquentiels.

5.1 L'univers en mémoire multi-niveaux

La première étape dans la parallélisation de la simulation consiste en la création d'un univers non plus local mais stocké en mémoire multi-niveaux. La définition d'un tel univers ML_UNIVERS se fait simplement en héritant de la classe UNIVERS, qui représente la spécification abstraite de l'univers, ainsi que d'une structure de stockage en mémoire multi-niveaux.

Les corps étant crées en mémoire partagée, l'univers ne contient plus des objets de type BODY mais de type SHD_BODY. Comme la classe SHD_BODY hérite de BODY, il n'y a pas besoin, grâce au polymorphisme, de redéfinir les opérations effectuées sur des objets de type BODY.

5.2 Parallélisation de la simulation

Une fois la distribution de l'univers achevée, la parallélisation de la simulation des n-corps se fait en deux étapes.

La première étape consiste à définir de nouveaux opérateurs parallèles qui remplaceront ceux séquentiels. Dans le cas de la simulation des n-corps, les opérateurs étant relativement simples leur parallélisation se fait tout simplement en héritant à la fois de l'opérateur séquentiel et de la classe abstraite PARALLEL_OPERATOR.

La deuxième étape concerne l'utilisation des ces opérateurs. Afin de fonctionner de manière parallèle, il faut en effet les utiliser avec les itérateurs appropriés. Cette utilisation dépend ainsi du fait que les opérateurs font intervenir des éléments locaux ou distants. Nous allons donc examiner comment s'effectue la parallélisation des différents opérateurs de la simulation.

5.2.1 Parallélisation de la mise à jour des corps

Cet opérateur est chargé de mettre à jour les corps stockés dans l'univers. Lors d'une exécution parallèle, chaque processeur devra alors uniquement mettre à jour ses éléments locaux. La définition de la classe PARALLEL_UPDATOR se fait simplement par héritage de l'opérateur séquentiel UPDATOR et de l'abstraction PARALLEL_OPERATOR qui prend en charge les problèmes de synchronisation.

L'utilisation de l'opérateur UPDATOR se fait alors simplement en remplaçant l'ancienne phase d'initialisation :

!!updator.make(universe.items, slice)

par :

!!updator.make(universe.local_items, slice)

Les opérations de mise à jour définies dans l'opérateur séquentiel ne se font ainsi sur les corps dont le processeur est responsable et non plus sur l'ensemble des corps stockés dans l'univers.

5.2.2 Parallélisation du calcul des contributions

Dans le calcul des interactions entre les corps, seul l'opérateur UNIV_CONTRIBUTOR nécessite d'être parallélisé. En effet, l'opérateur BODY_CONTRIBUTOR n'est utilisé que comme opération effectuée par la méthode *forall_operation* de BODY_CONTRIBUTOR et est donc utilisé de façon séquentielle par chaque processeur.

La version parallèle de UNIV_CONTRIBUTOR est définie uniquement par héritage de la version séquentielle ainsi que de l'abstraction PARALLEL_OPERATOR. La figure 5.2.2 montre les relations entre les différentes classes.

Lors de son initialisation, la classe UNIV_CONTRIBUTOR nécessite deux itérateurs. Il calcule alors la contribution de l'ensemble des masses fournies par le premier itérateur sur l'ensemble des corps fournis par le second itérateur. Pour chaque processeur, le UNIV_CONTRIBUTOR parallèle calcule toujours la contribution de tout l'univers. En revanche, il ne calcule cette contribution que sur un sous-ensemble de l'univers. Le parallélisme provient ainsi du fait que pour chaque processeur, on effectue le calcul de la contribution de l'ensemble de l'univers uniquement sur les corps appartenant à ce processeur, comme le montre la figure 5.2.2. L'initialisation du PARALLEL_UNIV_CONTRIBUTOR se fait donc en remplaçant la phase d'initialisation séquentielle :

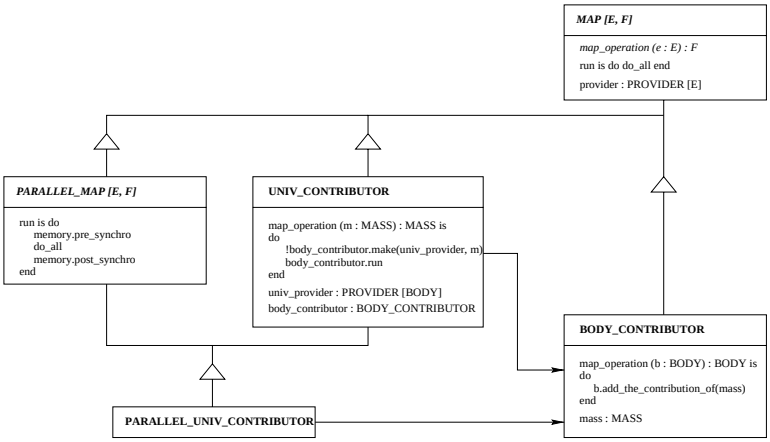

FIG. 2 – *Schéma de parallélisation de la contribution*

!!univ_contributor.make(universe.items, universe.items)

par :

!!univ_contributor.make(universe.items, universe.local_items)

L'itérateur sur l'ensemble des masses est donc inchangé et parcourt l'ensemble de l'univers tandis que l'itérateur sur les corps retourne uniquement les corps dont le processeur est responsable.

5.3 La version parallèle de la simulation des n-corps

Une fois que tous les composants parallèles décrits dans les paragraphes précédents ont été écrits, la parallélisation de la simulation se fait unique-

ment en remplaçant les collections et opérateurs séquentiels par leurs équivalents parallèles. Les modifications sont ainsi minimes et la majeur partie du code de la simulation est réutilisée.

6 Conclusion

Nous avons montré comment une approche résolument orientée objet et fondée sur l'utilisation de patrons de conceptions spécifiques permet d'exprimer des traitements à un niveau où l'irrégularité des structures sous-jacentes est abstraite; ce qui permet de les exécuter optimalement tant sur des architectures centralisées que parallèles à mémoire (virtuellement) partagée ou distribuée, ou même mixtes (méta-computing). Cette présentation s'est appuyée sur l'exemple de l'environnement EPEE, qui offre un cadre de conception, basé sur le langage Eiffel, pour développer des composants logiciels réutilisables pour applications traitant de grandes quantités d'événements, de données ou de calculs.

Remerciements

L'étude de cas des N corps a été développée et rédigée par Thomas Leseney pendant son stage de DEA (Ifsic, 1997).

Références

[1] America (P.). – Pool-T: A parallel object-oriented programming. *In : Object-Oriented Concurrent Programming*, éd. par Yonezawa (A.). pp. 199–220. – The MIT Press, 1987.

[2] Andersen (B.). – Ellie language definition report. *Sigplan Notices*, vol. 25, n° 11, November 1990, pp. 45–64.

[3] Bennett (J. K.). – The design and implementation of DistributedSmalltalk. *In : OOPSLA'87 Proceedings*. – October 1987.

[4] Caromel (D.). – Towards a method of object-oriented concurrent programming. *Communications of the ACM*, vol. 36, n° 9, September 1993, pp. 90–102.

[5] Chandra (R.), Gupta (A.) et Hennessy (J. L.). – Cool: a language for parallel programming. *In : Languages and Compilers for Parallel Computing*, éd. par et al. (D. G.). – MIT Press, 1990.

[6] Gamma (E.), Helm (R.), Johnson (R.) et Vlissides (J.). – *Design Patterns: Elements of Reusable Object-Oriented Software*. – Addison Wesley, 1994.

[7] Guidec (F.). – *Un cadre conceptuel pour la programmation par objets des architectures parallèles distribuées : application à l'algèbre linéaire*. – Thèse de doctorat, IFSIC / Université de Rennes 1, juin 1995.

[8] Guidec (F.), Jézéquel (J.-M.) et Pacherie (J.-L.). – An object oriented framework for supercomputing. *Journal of Systems and Software*, vol. Special Issue on *Software Engineering for Distributed Computing*, Juin 1996.

[9] Hoare (C. A. R.). – Communicating sequential processes. *Comm. of the ACM*, vol. 21, n° 8, August 1978, pp. 666–677.

[10] Jézéquel (J.-M.). – *Object Oriented Software Engineering with Eiffel*. – Addison-Wesley, Mars 1996. ISBN 1-201-63381-7.

[11] Jézéquel (J.-M.), Guidec (F.) et Hamelin (F.). – Parallelizing Object Oriented Software Through the Reuse of Parallel Components. *Object-Oriented Systems*, vol. 1, 1994, pp. 149–170.

[12] Jézéquel (J.-M.) et J.-L. (P.). – Operator design pattern, application to parallel computation. *In : Collected papers from the PLoP '96 and EuroPLoP '96 Conferences*. – Washington University Department of Computer Science, Fév. 1997.

[13] Jézéquel (J.-M.), Matsuoka (S.), Sato (N.) et Yonezawa (A.). – A methodology for specifying data distribution using only standard object-oriented features. *In : Proc. of International Conference on Supercomputing*. – Vienna, Austria, Juil. 1997.

[14] Jézéquel (J.-M.) et Pacherie (J.-L.). – *Shared Objects in KOOPE*. – Rapport technique n° 2.3, Irisa-Intel ERDP, Juin 1995.

[15] Jézéquel (J.-M.) et Pacherie (J.-L.). – Parallel Operators. *In : ECOOP'96 proceedings*, éd. par Cointe (P.). pp. 384–405. – Lecture Notes in Computer Science, Springer Verlag, Juil. 1996.

[16] Matsuoka (S.) et Yonezawa (A.). – Analysis of inheritance anomaly in object-oriented concurrent programming languages. *In : Research Directions in Concurrent Object Oriented Programming*, éd. par Agha (G.), Wegner (P.) et Yonezawa (A.). – MIT Press, 1993.

[17] Meseguer (J.). – Solving the inheritance anomaly in concurrent object-oriented programming. *In : Proceedings ECOOP'93*, éd. par Nierstrasz (O.). pp. 220–246. – Kaiserslautern, Germany, Juil. 1993.

[18] Meyer (B.). – Systematic concurrent object-oriented programming. *Communications of the ACM*, vol. 36, n° 9, September 1993.

[19] Pacherie (J.-L.). – Modèle et environnemnt de programmation parallèle par objets. *In : Huitièmes rencontres francophones du parallélisme, RenPar'8*, éd. par Castanet (R.) et Roman (J.). p. 9. – Bordeaux, France, Mai 1996.

[20] Yokote (Y.) et Tokoro (M.). – The design and implementation of ConcurrentSmalltalk. *In : OOPSLA'86 Proceedings*. – 1986.

[21] Yonezawa (A.), Briot (J.-P.) et Shibayama (E.). – Object-oriented concurrent programming in ABCL/1. *In : OOPSLA'86 Proceedings*. – September 1986.