# THÈSE

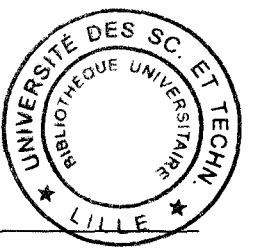
*pour l'obtention du titre de*

**DOCTEUR en INFORMATIQUE**

**à l'Université des Sciences et Technologies de Lille**

*par*

**Violeta FELEA**




# METHODOLOGIE DE CONCEPTION ET EXECUTION EFFICACE DE PROGRAMMES JAVA DISTRIBUES

Soutenue le : **15 mai 2003** devant la Commission d'examen

Jury :

| | | |
|---|---|---|
| **Président** | : | **Mme Laurence Duchien, Professeur à l'Université des Sciences et Technologies de Lille** |
| **Rapporteurs** | : | **Mme Brigitte Plateau, Professeur à l'Institut National Polytechnique de Grenoble** |
| | | **M. Denis Caromel, Professeur à l'Université de Nice, Sophia Antipolis** |
| **Examinateurs** | : | **M. Pierre Lecouffe, Maître de Conférences à l'Université des Sciences et Technologies de Lille** |
| | | **M. Dan Grigoras, Professeur à l'Université Al. I. Cuza, Iasi, Roumanie et à l'Université College Cork, Irelande** |
| **Directeur** | : | **M. Bernard Toursel, Professeur à l'Université des Sciences et Technologies de Lille** |

**UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE**
**Laboratoire d'Informatique Fondamentale de Lille — UMR 8022**
**U.F.R. d'I.E.E.A. – Bât. M3 – 59655 VILLENEUVE D'ASCQ CEDEX**
**Tél. : +33 (0)3 20 43 47 24 – Télécopie : +33 (0)3 20 43 65 66 – email : direction@lifl.fr**

# Remerciements

Je remercie Madame Laurence Duchien, Professeur à l'Université des Sciences et Technologies de Lille, pour m'avoir fait l'honneur de présider le jury de cette thèse.

Je tiens à remercier également Madame Brigitte Plateau, Professeur à l'Institut National Polytechnique de Grenoble et Monsieur Denis Caromel, Professeur à l'Université de Nice, Sophia Antipolis, pour avoir accepté de rapporter cette thèse.

Je remercie Monsieurs Pierre Lecouffe, Maître de Conférences à l'Université des Sciences et Technologies de Lille et Dan Grigoras, Professeur à l'Université "Al.I. Cuza", Iasi, Roumanie, d'avoir accepté de faire partie de mon jury.

Je remercie Monsieur Bernard Toursel, Professeur à l'Université des Sciences et Technologies de Lille, qui a dirigé mes travaux de recherche, pour m'avoir guidé au long de cette thèse. Ses conseils, son support et sa confiance m'ont aidée dans la réalisation de cette thèse.

J'exprime un grand merci à la première personne qui m'a ouvert le chemin de la recherche au Laboratoire d'Informatique Fondamentale de Lille, au Professeur Jean-Pierre Steen. Il m'a encouragé tout au long de l'année de DEA, et soutenu pour poursuivre mes études en thèse. Je le remercie d'avoir partagé une expérience riche d'enseignements, d'avoir été un exemple de générosité et de correctitude et un support moral inégalable. Un grand merci pour toutes ses relectures, commentaires de cette thèse, et pour toute sa disponibilité.

Je remercie également tous les membres (anciens et présents) de l'équipe PALOMA, au sein de laquelle j'ai fait les premiers pas dans la recherche. J'apprécie l'aide de tous ceux qui ont facilité le déroulement de mes expérimentations, à Pierre, Bruno, et à tous qui m'ont encouragée et soutenue.

Je souhaiterai remercier ma famille, pour leur aide et encouragements, même de loin. Leurs lettres, téléphones et messages électroniques sont toujours arrivés au bon moment et leurs conseils m'ont aidée à traverser les moments difficiles. J'ai une pensée particulière pour tous ceux qui ne m'auraient pas vue docteur, et à qui cela aurait fait tant plaisir.

Je remercie finalement mes ami(e)s, de loin et de proche, qui n'ont pas toujours compris tout ce que je faisais, mais ont supporté mes divagations. Nos promenades, nos discussions et nos crêpes ensemble m'ont donnée la force d'aller au bout de ce travail.

Un dernier merci pour l'ambiance qui règne à Lille et ses alentours, cela m'a permis de me rafraîchir les idées et m'a laissée comprendre l'importance de la vie.

# Table des matières

# Table des figures

# Première partie

# Environnements de développement et d'exécution d'applications distribuées et parallèles

# Chapitre 1

# ADAJ : Applications Distribuées Adaptatives en Java

L'objet de cette thèse est de concevoir un environnement de programmation et d'exécution pour les applications parallèles et distribuées qui en facilite la conception et optimise l'exécution.

Le projet proposant cet environnement s'appelle ADAJ (Applications Distribuées Adaptatives en Java).

Ce premier chapitre a pour but de présenter les problèmes spécifiques à ces applications, de poser les problèmes et de définir les objectifs à atteindre.

## 1.1 Contexte

L'augmentation exponentielle des performances des processeurs élémentaires, la demande en puissance de traitement et en capacité de mémorisation et de communication d'un nombre important d'applications nécessitent la considération des environnements d'exécution distribués, multiprocesseurs.

Les architectures composées de multiples processeurs interconnectés sont classées en deux grandes catégories :

- les systèmes fortement couplés (*tightly coupled systems*) correspondent à une architecture parallèle organisée autour d'une mémoire commune partagée par les processeurs. Généralement, ces systèmes disposent de réseaux d'interconnexion internes performants.
- les systèmes faiblement couplés (*loosely coupled systems*) consistent en un ensemble de processeurs interconnectés par un réseau de communication et qui ne partagent pas de mémoire, chaque processeur ayant une mémoire locale propre. Les processeurs sont connectés par des réseaux lents par rapport aux performances offertes par les processeurs.

Un système multiprocesseur trouve son utilité dans l'amélioration à la fois, des performances en terme de vitesse d'exécution, et de la sécurité de fonctionnement.

Dans le domaine du calcul de hautes performances, la tendance actuelle évolue vers l'utilisation d'architectures multiprocesseurs, qui sont soit des machines parallèles, soit des réseaux de stations de travail. Les multiprocesseurs à mémoire partagée offrent une puissance de calcul importante, mais les accès à une mémoire partagée sont limités. Les réseaux de stations, appelés aussi *grappes* (*clusters*), sont des architectures à mémoire distribuée qui permettent l'utilisation des ressources (logicielles, matérielles) là où elles se trouvent. L'utilisation des grappes [BB99], dans des metacalculs (*meta computing*), est devenue une bonne alternative aux autres moyens de calcul, étant

de moindre coût, grâce à leur conception matérielle plus aisée ou à leur extension plus facile. A un niveau supérieur, des architectures hiérarchiques, les *grappes de grappes* (*grid*), font leur apparition de plus en plus. L'exploitation des ressources offertes consiste à recourir à des calculs sur la grille (*grid computing* [FK00]). Les systèmes distribués à grande échelle concernent les réseaux d'interconnexion usuels d'ordinateurs indépendants. Sur ces systèmes, pouvant comprendre des centaines de machines, s'effectue de *global computing*.

La grappe de stations de travail, la grille ou les réseaux à grande échelle réunissent des stations physiquement dispersées et interconnectées par le réseau. L'aspect qui caractérise ces architectures est l'hétérogénéité. L'hétérogénéité se décline sous trois formes :

- hétérogénéité du réseau (bande passante),
- hétérogénéité des ressources (puissance CPU[1], capacité mémoire),
- hétérogénéité du système d'exploitation des stations.

A part la gestion délicate de l'hétérogénéité, le problème de l'utilisation de ce type d'architecture est celui d'accès aux ressources et du partage des ressources entre les utilisateurs. Des outils sont disponibles qui donnent l'illusion d'un système unifié (*Single System Image*), à l'aide d'une mémoire distribuée partagée (*Distributed Shared Memory*), d'un système de gestion des fichiers distribués (*Network File System*), d'une gestion globale des ressources et d'ordonnancement, ou bien, au niveau des environnements de programmation, à l'aide des bibliothèques de communication.

## 1.2 Problèmes

Les réseaux hétérogènes soulèvent deux types de problèmes pour les applications. Un premier aspect concerne la transparence : les applications devraient être conçues le plus indépendamment possible de l'hétérogénéité des ressources disponibles. Le deuxième aspect concerne l'exécution, qui devrait prendre en compte les fluctuations dans la disponibilité des ressources. Les différences dans les capacités des ressources résultent d'une part, de divers systèmes utilisés et d'autre part, d'un partage avec d'autres utilisateurs de l'ensemble. L'efficacité de l'exécution peut être améliorée si et seulement si ces caractéristiques sont prises en compte.

Etant reconnu comme un langage portable qui masque la gestion réseau, le langage Java convient pour le développement et le déploiement d'applications dans un environnement hétérogène.

Java offre également un modèle d'objets Java distribués. Celui-ci est basé sur le protocole RMI [SUNb] (Remote Method Invocation), mécanisme d'invocation à distance des méthodes décrit dans le chapitre 2. Le langage Java gère l'hétérogénéité d'un environnement distribué, au niveau conceptuel des applications. La conception des applications distribuées et parallèles en Java est, donc, indépendante des variations possibles de la charge et de la performance des machines ou dans le débit du réseau.

A part la portabilité et le modèle d'objets distribués, le langage Java intègre tous les mécanismes nécessaires à un langage orienté objets, dans une interface de programmation très développée. Il offre des outils pour la programmation concurrente et de sécurité d'exécution. A travers toutes ses spécificités, le langage Java permet de la programmation sûre et simple des applications sur un nombre de plus en plus grand d'environnements de travail (Windows, Linux, Solaris, MacOS). La portabilité du langage entraîne cependant un ralentissement par rapport aux programmes écrits dans d'autres langages (spécialement en C++). Ce qui est gagné en portabilité, est perdu, donc, en partie, en rapidité d'exécution.

[1]Central Processing Unit

Deux voies possibles ont été proposées pour améliorer l'efficacité de l'exécution :
- en se basant sur l'environnement distribué Java existant, Java RMI, offrir des mécanismes d'optimisation (de la communication par exemple - [NPH99, MNV+99]), ou remplacer le protocole de communication par d'autres plus efficaces,
- en étendant l'environnement Java RMI, offrir des bibliothèques de primitives ou des mécanismes d'optimisation de l'exécution.

Dans le deuxième cas, une classification des outils peut être faite en fonction du niveau de l'intervention des mécanismes :
- au niveau applicatif, par des outils d'expression du parallélisme (en donnant en même temps une orientation vers une méthodologie de programmation), ou/et par des mécanismes de placement explicite d'objets,
- au niveau exécutif, par des mécanismes de placement implicite d'objets ou par des mécanismes de redistribution d'objets.

### 1.2.1 Efficacité au niveau applicatif

Au niveau applicatif, différentes approches ont essayé de donner des réponses aux problèmes d'efficacité d'exécution des applications distribuées, qui se basent sur le parallélisme.

Dans la recherche en programmation parallèle, deux axes se distinguent : une voie compilation, qui se base sur la parallélisation automatique et une voie bibliothèque qui concerne la parallélisation manuelle, orientée par l'utilisateur.

Dans l'extraction automatique du parallélisme, en partant du code séquentiel, des entités parallèles sont générées, de manière transparente pour l'utilisateur. En langage Java, au niveau du développement, on ne dispose pas d'outils de parallélisation automatique, même si les outils de base (les threads Java) sont fournis.

Le deuxième axe concerne les bibliothèques. Deux grandes approches sont abordées : le parallélisme de tâches et le parallélisme de données.

Le parallélisme de tâches consiste à identifier des tâches et à les affecter à des processeurs distincts. A cette forme de parallélisme est souvent associé le modèle de programmation MIMD (Multiple Instruction Multiple Data). Une caractéristique fondamentale est le côté fortement asynchrone de ce type de parallélisme, dès que des tâches indépendantes s'exécutent simultanément.

Par l'asynchronisme des appels, les projets ProActive [BCHV00], Ajents [ICB99] ou Do ! [LP00] expriment un volet du parallélisme, le parallélisme de tâches.

Le parallélisme de données est orienté vers des ensembles importants de données sur lesquels des traitements identiques sont appliqués. Les ensembles de données sont distribués aux différents processeurs, chacun appliquant alors le traitement sur son sous-ensemble de données. A cette forme de parallélisme est souvent associé le modèle de programmation SPMD (Single Program Multiple Data) qui fait correspondre à la structure régulière des données du programme une structure régulière du traitement.

Cette expression du parallélisme est présente dans la bibliothèque DPJ [IGD+97].

Dans ces deux expressions du parallélisme, des problèmes de placement des données et de gestion des communications surgissent. Par des mécanismes de placement explicite d'objets, le programmeur peut diriger le déploiement de l'application en fonction des nécessités. Le placement explicite peut répondre à des demandes initiales de l'application, mais il ne résout pas les contraintes d'hétérogénéité du système ou de variations dans l'exécution de l'application.

L'aspect communication est lié au placement des objets concernant la transparence de l'appel

qui exige qu'une communication, entre des entités situées dans des espaces d'adressage différents, soit semblable à une communication locale.

### 1.2.2 Efficacité au niveau exécutif

L'efficacité de l'exécution d'une application distribuée et parallèle est directement liée à un usage optimal des ressources. Pour les applications orientées objets, les consommateurs de ressources sont les objets. Le déploiement d'une application, c'est-à-dire le placement des objets, génère une consommation des ressources de la machine les hébergeant.

Un problème important pour le déploiement d'une application répartie sur un environnement hétérogène est, donc, la distribution des objets. DPJ [IGD+97] offre une bibliothèque pour exprimer la distribution des objets (en restant explicite lors de leur création), aspect qui est autant lié à la conception des programmes qu'à l'efficacité de l'exécution.

La transparence de développement et de déploiement d'une application parallèle et distribuée reste une tâche significative, vue la difficulté de la mise en œuvre. Les différentes approches existantes proposent une distribution explicite des objets, soit statique (projets Ajents ou Do!) soit dynamique (en ajoutant des facilités de migration - projets ProActive ou JavaParty [PZ97]).

Le modèle d'objets distribués Java (Java RMI) permet aux concepteurs de programmes de distribuer une application, mais n'offre aucun mécanisme pour assurer l'adaptabilité de l'application distribuée pendant l'exécution. La structure d'une application Java répartie est figée, en demandant du travail additionnel de la part du concepteur : le découpage d'une application, en parties client et serveur, est décidé au moment de la conception et les différentes parties sont lancées manuellement, sans qu'elles puissent être redistribuées pendant l'exécution. Ces restrictions empêchent l'adaptabilité de l'exécution, en vue d'une meilleure efficacité.

Les algorithmes d'équilibrage de charge dans les couches *middleware* (*intergicielles*) offrent de meilleures performances de l'exécution d'une application dans un environnement hétérogène. Ces algorithmes devraient considérer une double fluctuation : dans les puissances des machines et dans l'évolution de l'application.

Aucun des projets cités précédemment, qui sont détaillés dans la section 5, n'offre de mécanisme d'adaptation automatique des applications aux évolutions des calculs ou aux changements dans l'environnement d'exécution, l'efficacité d'exécution des applications étant réalisée seulement au niveau applicatif. Seuls les outils de base, notamment la migration des objets, sont fournis, par les projets JavaParty ou Ajents, et dans une version étendue de ProActive.

L'adaptation automatique des applications est exigée par l'apparition de déséquilibres dans l'utilisation des ressources : certaines sont utilisées de manière intense pendant que d'autres ne le sont pas du tout.

Pour les applications parallèles, la source de déséquilibre peut être à la fois interne et externe.

- Le déséquilibre interne apparaît si des quantités de traitement liées à l'application ne sont pas uniformément distribuées.
- Le déséquilibre externe est le résultat du partage de l'unité centrale et de la mémoire avec d'autres processus de la station.

Basé sur une bibliothèque de classes C++ et utilisant PVM[2] pour le contrôle des processus et des communications, Dome [ABL+95] propose un équilibrage de charge basé sur l'observation de la vitesse du processeur et de la performance de communication. Les applications visées en Dome, de type SPMD, favorisent, du fait des synchronisations, des prises de décisions sur le nouveau

[2]Parallel Virtual Machine

placement des données en fonction du temps d'exécution de la dernière phase de calcul. Les interdépendances qui peuvent exister entre les données des applications sont peu influentes.

## 1.3 Objectifs ADAJ

La conception d'une application parallèle distribuée et son exécution efficace sont des aspects étudiés surtout dans le contexte des réseaux hétérogènes, dynamiques et de grande taille. Rendre la conception plus aisée et la plus indépendante possible du support d'exécution, assurer une distribution automatisée et optimisée de l'application sur la plate-forme d'exécution sont des problèmes soulevés qui sont partiellement résolus.

Les applications visées sont dynamiques et irrégulières. Par rapport aux algorithmes statiques dont l'exécution est connue par avance et ne change pas quelles que soient les données d'entrée, les algorithmes dynamiques sont caractérisés par une variation d'exécution de l'application selon les données d'entrée. L'irrégularité d'une application est liée au type du graphe de dépendance entre les objets de l'application. Pour le cas d'une application statique, le graphe est complètement prévisible. Lors d'une application irrégulière, le graphe est :

- sémi-prévisible : il est possible de structurer l'algorithme dans une succession d'étapes exécutées en parallèle (appelées *phases*). L'irrégularité dans ce cas est liée au nombre de phases ou au nombre de tâches créées dans chaque phase.
- imprévisible : le graphe est corrélé aux données et ne peut être construit que lors de l'exécution des tâches. La même situation se retrouve dans les applications coopératives où le nombre d'acteurs et leurs actions peuvent varier de façon imprévisible. Dans ce cas, l'irrégularité est difficile à maîtriser.

Les solutions envisagées pour une conception aisée, et une exécution efficace, reposent sur l'introduction, au niveau applicatif, des frameworks (cadres logiciels) ou bibliothèques de développement et, au niveau des middlewares, des mécanismes qui peuvent dynamiquement adapter aux modifications du support d'exécution et aux évolutions des calculs, la granularité des traitements, le degré de parallélisme et la distribution.

Les objectifs que nous nous sommes fixés, dans le cadre d'ADAJ (Applications Distribuées Adaptatives en Java) sont les suivants :

- simplifier le travail du programmeur en cachant des problèmes liés à la gestion du parallélisme, en fournissant une interface de programmation (API[3]) complète pour que le programmeur puisse mener facilement le développement des applications parallèles,
- faciliter le développement des applications et permettre leur déploiement automatique ou quasi-automatique dans des environnements a priori hétérogènes, de manière la plus transparente possible pour l'utilisateur,
- assurer une exécution efficace des traitements, par des mécanismes d'équilibrage de charge inter et intra-applications.

ADAJ est ainsi conçu comme un environnement de programmation et de déploiement d'applications Java distribuées et parallèles.

Dans le contexte complexe des applications irrégulières et dynamiques, c'est-à-dire dont le comportement est imprévisible, qui s'exécutent dans un environnement distribué, ADAJ offre des outils qui permettent de bénéficier de l'existence de plusieurs processeurs. L'utilisation des opérations parallèles peut fournir une grande efficacité et augmenter la performance d'une application Java dont

[3]Application Program(ming) Interface

les traitements sont décomposables explicitement en sous-traitements indépendants.

Les algorithmes parallèles s'appuient soit sur le parallélisme de données, qui consiste à découper les données et les traiter en parallèle selon leur placement sur les différents processeurs, soit sur le parallélisme de tâches qui consiste à exécuter en parallèle différentes tâches. ADAJ cherche à faciliter la conception d'un programme parallèle en offrant une expression simple du parallélisme. Le concept de collection distribuée qu'ADAJ propose marie les deux modèles de programmation parallèle. Ainsi, l'utilisateur a la possibilité d'activer des traitements sur des objets fragmentés et distribués, en privilégiant à la fois le parallélisme de l'application (sur les objets) et la distribution pour la régulation de charge, mariant les deux techniques d'obtention de l'efficacité, au niveau applicatif et d'exécution.

Nos objectifs ne sont pas de faire de l'extraction automatique du parallélisme, mais de proposer un modèle de programmation parallèle, apportant un maximum de transparence, à la fois pour simplifier le travail du programmeur (en cachant les problèmes complexes dus à l'écriture d'un programme parallèle) et pour améliorer l'efficacité d'exécution des applications.

Conçu comme un support d'exécution d'applications orientées objets parallèles et distribuées, ADAJ est une plate-forme s'exécutant dans un environnement multi-utilisateurs, multi-applications pour un ensemble de stations de travail, interconnectées par un réseau, stations capables d'accueillir des machines virtuelles Java. Un premier objectif concernant la plate-forme d'exécution est la portabilité : conserver la machine virtuelle Java non-modifiée assure l'intégration de l'environnement sur toute machine disposant d'une distribution Java. ADAJ est un gestionnaire de placement et de migration chargé de répartir automatiquement les applications parallèles sur un ensemble de stations.

L'efficacité de l'exécution des programmes distribués et parallèles est obtenue, en ADAJ, non seulement à travers des outils conceptuels, mais aussi grâce aux mécanismes qui peuvent dynamiquement adapter l'exécution aux caractéristiques de l'environnement d'exécution et au comportement de l'application elle-même.

ADAJ vise à résoudre les deux types de déséquilibres à travers des mécanismes d'équilibrage intra et inter-applications. La recherche d'une exécution efficace des traitements distribués passe par l'introduction de mécanismes d'observation. Ces mécanismes permettent d'acquérir une connaissance du comportement du traitement et de la plate-forme support durant l'exécution. Ils s'appuient sur deux sources d'information : un dispositif d'observation des relations entre les objets distribués proposé par [BLL00], et un outil d'estimation de la charge et de son évolution [BOT01]. ADAJ adapte la distribution de charge à la fois aux variations du support d'exécution et à l'évolution des calculs et de la quantité de données traitées. Cette approche s'associe aux outils de conception proposés pour accroître l'efficacité de l'exécution des traitements Java répartis.

ADAJ est un environnement conçu au-dessus de JavaParty, héritant ainsi des améliorations apportées au modèle standard d'objets Java distribués de RMI. La thèse est structurée dans quatre parties : cette première partie présente le contexte du travail et les objectifs du projet ADAJ. Dans les chapitres suivants, nous décrivons brièvement l'environnement d'objets répartis Java RMI et l'environnement JavaParty, sur lesquels ADAJ est construit. La deuxième partie est focalisée sur l'approche de méthodologie de conception proposée en ADAJ, pour la programmation d'applications distribuées et parallèles. La troisième partie concerne le mécanisme d'équilibrage de charge introduit en ADAJ pour améliorer l'efficacité d'exécution de ses applications. Les conclusions générales et les perspectives font l'objet de la quatrième partie.

# Chapitre 2

# L'environnement distribué Java RMI

Une application Java utilise des objets situés dans un même espace d'adressage. Dans une application Java distribuée, les objets résident sur des machines distinctes du réseau. Dans le modèle local (dont les caractéristiques sont présentées dans la section 2.1), un objet est sollicité par un appel de méthode dans la même machine virtuelle JVM (Java Virtual Machine) [1]. Au contraire, dans un environnement distribué, l'appel de méthode transite, via le réseau, pour être appliqué sur l'objet, là où il réside. Pour le programme appelant de telles méthodes, les objets auxquels elles s'appliquent sont appelés *objets distants.*

La non-localité des objets distants est partiellement masquée par une couche intergicielle, appelée *middleware.* Elle se charge de rendre transparents, pour le développeur, les appels de méthodes sur des objets distants. Cette couche intervient, dans le modèle OSI[2], au niveau des couches session et présentation.

Deux types de middleware sont les plus utilisés, CORBA (Common Object Request Broker Architecture) [OMG] et RMI (Remote Method Invocation) [SUNb], le dernier sera présenté plus en détail dans les sections 2.2, 2.3 et 2.4.

## 2.1 Le langage Java

Java est un langage qui intègre des concepts de la programmation orientée objets pour offrir des caractéristiques de simplicité, portabilité, sûreté, multithreading et distribution.

La simplicité est essentiellement réalisée grâce à l'inexistence d'une arithmétique sur les pointeurs comme dans le langage C/C++. La notion de *référence* masque la notion de pointeur, comme moyen d'accéder à un objet. Un objet est créé en instanciant une classe. Une classe encapsule l'initialisation des objets de cette classe via des constructeurs, ainsi que des variables et méthodes liées à la classe et non à ses instances. Ces attributs et méthodes de classe sont appelés *statiques*, puisqu'ils existent indépendamment de toute instance et sont partagés par toutes les instances de la classe. La simplicité provient également de l'interdiction d'usage d'un héritage multiple des classes. En revanche, la notion d'interface permet de simuler l'héritage multiple.

Le langage Java offre de l'indépendance vis-à-vis de l'architecture par le code généré qui ne dépend pas de la plate-forme d'exécution. Le compilateur fournit du code proche du langage machine, appelé *bytecode*, qui est

- indépendant de la plate-forme,

[1] JVM = machine virtuelle Java

[2] Open Systems Interconnect

– interprétable dans une machine virtuelle.

Ainsi, un programme Java peut être exécuté dans tout environnement disposant d'une JVM. Etant interprété, le langage Java permet la portabilité des applications.

Le typage fort, ainsi que le mécanisme d'exceptions rend le langage Java sûr. Les conversions implicites en nombre limité, l'interdiction de définir d'autres opérateurs de conversion, la vérification à l'exécution du bytecode sont des aspects qui assurent le programmeur d'un code correct. Le mécanisme d'exceptions permet de séparer le flot de contrôle classique du flot de contrôle lié à un comportement exceptionnel lié à une erreur.

Un programme Java peut exécuter plusieurs tâches simultanément, grâce à l'aspect multithreading. Ce mécanisme de gestion de plusieurs threads est lié au langage, intégrant en même temps des mécanismes de synchronisation.

Java offre aussi un modèle d'objets répartis sur plusieurs JVM, qui communiquent grâce au protocole RMI, présenté dans les sections suivantes.

## 2.2 Le mécanisme RMI

Le protocole RMI permet de créer des objets distants dont les méthodes peuvent être appelées à distance.

Schématiquement, le mécanisme RMI permet, sur une machine appelée *machine serveur*, de créer un objet et de le rendre accessible aux autres machines, pour lesquelles il devient un objet distant. Sur l'une de ces machines, dite *machine cliente*, une application Java peut alors récupérer, localement, une représentation de l'objet distant, appelée *talon (stub)*. Elle utilise ce talon comme elle utiliserait un objet local sur la machine cliente. Notamment, elle peut appeler, sur ce talon, les méthodes d'instance définies dans la classe de l'objet distant. Cet appel local de méthode sur un talon de l'objet distant est alors transmis, via le réseau, depuis la machine cliente jusqu'à la machine serveur où il engendre l'exécution, sur l'objet distant, de la méthode correspondant à celle qui a été appelée à distance.

Les objets statiques sont considérés uniques dans un programme centralisé, et ainsi, les méthodes statiques sont comme des méthodes uniques. Le même principe s'applique pour les attributs statiques d'une classe. Dans le cas distribué, les classes sont dupliquées implicitement sur toutes les machines et les attributs et méthodes statiques ne sont plus uniques dans l'environnement. En conséquence, la sémantique de la partie statique interdit l'usage des attributs statiques ou l'appel des méthodes statiques en RMI.

### 2.2.1 Le principe RMI

Le principe du mécanisme RMI, similaire à son ancêtre RPC (Remote Procedure Call) qui permet d'appeler des fonctions sur un système distant, repose sur une analogie entre le modèle client-serveur et l'appel de fonctions dans les langages de programmation. Les différentes étapes lors d'un appel de méthode, comparé à un appel de fonction, sont :

- la connexion d'un client qui correspond à l'appel de fonction,
- la requête qui correspond au passage des paramètres,
- le traitement du service qui correspond à l'exécution du corps de la fonction,
- la réponse du serveur qui correspond à la valeur de retour de la fonction.

### 2.2.2 Le fonctionnement RMI

L'implémentation de RMI est basée sur les notions de *talon* (*stub*) et de *squelette* (*skeleton*). Les étapes de l'invocation d'une méthode sur un objet distant sont représentées dans la figure 2.1 ([RD00]).

Pour que l'objet distant soit rendu accessible à distance, il est *exposé*, tout d'abord, dans un serveur d'objets (aspect réalisé de manière implicite, par l'extension de la classe *java.rmi.server.UnicastRemoteObject*) (opération 1, appelée *exposition d'objet distant*). L'exposition de l'objet consiste à le rendre accessible dans le serveur d'objets. Ensuite, pour pouvoir être localisé, il est inscrit dans un serveur de noms[3] qui gère une table d'associations de références et de noms (opération 2, appelée *publication d'objet distant*).

Du côté client, le talon, correspondant à l'objet distant, est récupéré par une interrogation du serveur de noms (opérations 3 et 4). Le talon simule l'objet distant : il offre les mêmes fonctionnalités que l'objet distant. Lors de l'appel d'une méthode sur le talon (opération 5), ce dernier se connecte sur le serveur hébergeant l'objet distant et émet ensuite vers ce serveur une suite d'octets comprenant les identificateurs de l'objet distant et de la méthode appelée, suivis des arguments sérialisés (opération 6). Cette opération est appelée *marshalling*.

Du côté serveur, une fois l'objet localisé par le serveur, son squelette se charge de désérialiser les arguments et d'appeler la méthode souhaitée : cette opération est appelée *unmarshalling* (opération 7). Le squelette simule un appel local du côté serveur et engendre l'exécution de la méthode sur l'objet. Une fois la méthode exécutée, c'est le squelette qui reçoit le résultat (opération 8). Il se charge de le retourner en direction du talon qui a transféré l'appel. Cette transmission est réalisée par marshalling et la suite d'octets transmise contient la forme du résultat, valeur ou exception, suivi de sa valeur sérialisée (opération 9). Finalement, le talon reçoit ce résultat qu'il reconstruit (par unmarshalling) avant de le retourner au client (opération 10).

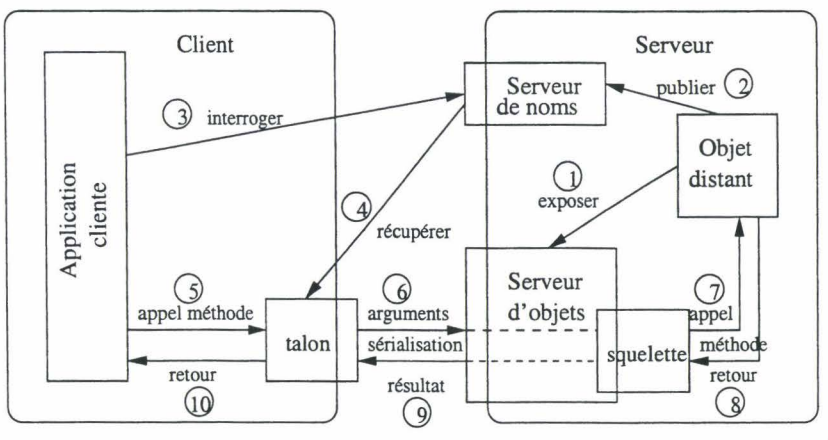



FIG. 2.1 – Le schéma de fonctionnement d'un appel distant Java RMI

[3] le serveur de noms est lancé par l'outil Java *rmiregistry*

## 2.3 L'environnement de programmation RMI

### 2.3.1 Méthodologie de programmation

Les outils de programmation offerts par le modèle d'objets répartis Java RMI imposent un style particulier de programmation, respectant les étapes suivantes :

- déclaration des interfaces des objets distants, pour rendre accessibles des services de l'objet distant,
- implémentation des interfaces des objets distants, pour définir les fonctionnalités des objets distants,
- définition d'une application serveur accueillant les objets distants (création et publication des objets distants),
- définition d'une application cliente en utilisant les objets distants.

L'interface de l'objet distant doit être connue par le client et le serveur, contenant la définition des services accessibles à distance. Les fonctionnalités de l'objet distant sont implémentées du côté serveur, ainsi que l'application serveur. Le client définit l'application qui utilise les objets distants.

#### Exemple

#### Définition de l'interface de l'objet distant

```
public interface AIntf extends java.rmi.Remote {
  public void mVoid(...) throws java.rmi.RemoteException;
  public int mReturn(...) throws java.rmi.RemoteException;
}
```

#### Définition de la classe de l'objet distant

```
public class A extends java.rmi.server.UnicastRemoteObject
               implements AIntf {
  public void mVoid(...) throws java.rmi.RemoteException {...}
  public int mReturn(...) throws java.rmi.RemoteException {...}
}
```

#### Définition de l'application serveur

```
public class Serveur {
  public static void main(String[] args) {
    // installer une sécurité réseau
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new java.rmi.RMISecurityManager());
    }

    try {
       AIntf obj = new A();

       // publier cet objet sous le nom "AServeur"
       java.rmi.Naming.bind("AServeur", obj);

       System.out.println("AServeur enregistré dans le registre");
```

```
    } catch (Exception e) {
        System.out.println("AServeur erreur : " + e.getMessage());
        e.printStackTrace();
    }
  }
}
```

**Définition de l'application cliente**

```
public class Client {
  public static void main(String[] args) {
     // créer et installer une sécurité
     if (System.getSecurityManager() == null) {
        System.setSecurityManager(new java.rmi.RMISecurityManager());
     }
     // "obj" est l'identification de l'objet qui réfère
     // l'objet distant sous le nom "AServeur"

     AIntf obj = null;

     try {
        // recherche de l'objet distant, dont le nom est "AServeur",
        // dans le serveur de noms lancé sur la machine args[0]
        obj = (AIntf)java.rmi.Naming.lookup("//" + args[0] + "/AServeur");
        System.out.println(obj.mReturn(...));
     } catch (Exception e) {
        System.out.println("AClient exception : " + e.getMessage());
        e.printStackTrace();
     }
  }
}
```

### 2.3.2 Style de programmation : contraintes

Ce style de programmation permet de créer des objets sur une machine distante et d'interagir avec eux à distance ; toutefois, il présente certaines contraintes :

- tous les objets distants doivent fournir une interface, qui étend l'interface *java.rmi.Remote* de Java et qui définit toutes les méthodes qui peuvent être appelées sur ces objets,
- pour pouvoir exposer l'objet distant, l'implémentation de l'interface distante correspondante doit étendre la classe Java, *java.rmi.server.UnicastRemoteObject*[4] risque d'imposer une restructuration du graphe des classes, parce que Java ne permet pas d'héritage multiple des classes,
- seuls les objets sérialisables peuvent être passés en paramètre ou comme retour d'une valeur d'une méthode distante ; donc les classes de ces objets doivent implémenter l'interface *java.io.Serializable*,
- une exception supplémentaire (*java.rmi.RemoteException*) doit être prévue pour les méthodes des objets distants (ainsi, une interface existante devrait être changée pour pouvoir l'appliquer

[4]ou implémenter la classe java.rmi.activation.Activatable

à un environnement distribué) ; à cause de cet aspect, l'appel à distance n'est pas transparent, et on dit que Java RMI fait la distinction entre les objets locaux (Java classiques) et les objets distants,
- les attributs d'instances ne peuvent être accessibles qu'au travers des méthodes,
- les méthodes statiques des classes distantes ne peuvent pas être invoquées à distance et leurs attributs statiques ne sont pas accessibles à travers la classe,
- les appels de méthodes distantes en RMI sont bloquants (synchrones), l'appelant étant en attente de retour de l'appel avant de poursuivre l'exécution avec l'appel suivant,
- les objets distants ont une localisation fixe,
- toutes les JVM doivent installer *java.rmi.RMISecurityManager*, pour la sécurité contre les talons distants, de façon à les empêcher d'accéder aux ressources locales quand ils sont chargés par le réseau sur l'environnement Java local.

## 2.4 L'environnement d'exécution

L'environnement d'exécution Java RMI est destiné à des machines virtuelles s'exécutant sur des stations connectées en réseau. La publication des objets distants est conditionnée par la présence d'un serveur de noms, opération réalisée grâce à l'outil *rmiregistry* du langage Java. Cette application doit s'exécuter en permanence durant l'exécution du programme, pour assurer l'accessibilité à distance aux objets.

# Chapitre 3

# L'environnement distribué JavaParty

JavaParty est un environnement de programmation et d'exécution pour les applications Java sur les grappes de stations accueillant des machines virtuelles Java.

L'environnement de programmation JavaParty permet de transformer un programme Java dans un programme distribué, en identifiant les objets nécessaires au déploiement de l'application sur les machines virtuelles de l'environnement distribué. L'identification est réalisée grâce au mot clé *remote* introduit dans le langage. Ces objets sont appelés *remote*. La compatibilité avec le langage Java est réalisée grâce à un compilateur spécifique.

JavaParty offre un espace d'adressage partagé, permettant aux objets remote d'être placés dans des machines virtuelles Java différentes. En palliant les inconvénients du modèle d'objets répartis RMI, JavaParty cache les mécanismes d'adressage et de communication à l'utilisateur et traite de manière interne les exceptions réseau. Ainsi, le programmeur n'est pas censé concevoir ni implémenter des protocoles de communication explicites.

## 3.1 L'environnement de programmation

JavaParty introduit dans son environnement de programmation une nouvelle sémantique d'objets remote, appartenant à une classe distribuée (*classe remote*). Une classe distribuée est la transposition d'une classe Java dans un environnement distribué. Ces instances, les objets remote, ont deux caractéristiques principales :

- elles sont accessibles à distance (à partir de tout l'environnement distribué),
- elles sont migrables (peuvent se déplacer d'une machine virtuelle à une autre, en acheminemant les appels de méthodes).

Le modèle d'objets JavaParty retient deux types d'objets dans l'environnement : les objets remote et les objets locaux, détaillés par rapport à leur sémantique, création et accès, gestion de la partie statique, passage des paramètres et mobilité.

### 3.1.1 Caractéristiques des objets remote

**Sémantique.** Les objets remote sont accessibles depuis tout l'environnement JavaParty sans les exporter explicitement ou les publier dans un service de noms comme en RMI.

**Création et accès.** La création et l'accès aux objets remote sont syntaxiquement similaires avec celles des classes Java. Les instances sont créées n'importe où dans l'environnement en utilisant le mot clé *new*.

Les méthodes ou attributs d'instance d'objets remote peuvent être accédés comme s'ils étaient des objets Java locaux. Il n'existe pas de traitement d'exceptions supplémentaires autres que celles déclarées par le programmeur.

**Partie statique.** Similaires aux classes Java, les classes remote ont aussi une représentation, à l'exécution, dans l'environnement distribué et sont accessibles au travers des constructions identiques à celles de Java.

Les classes remote, avec les variables et méthodes statiques, sont des entités uniques dans l'environnement, gérées par des objets remote, appelés *objets classes*. Les classes remote sont chargées une seule fois dans l'environnement distribué, donc un seul objet classe par classe remote est construit.

L'accès à la partie statique d'une classe remote peut être réalisé, en JavaParty, par le biais de l'objet classe, de manière transparente.

**Passage des paramètres.** Les objets remote, arguments ou résultats des invocations de méthodes, sont passés par référence.

**Mobilité.** Un objet remote peut migrer vers une machine virtuelle arbitraire de l'environnement d'exécution, les références précédentes peuvent encore être utilisées, grâce au mécanisme de *forwarding*. Le mécanisme de migration des objets remote est présenté dans la section 3.3.3. Tous les objets remote peuvent migrer sauf ceux déclarés appartenant aux classes résidentes.

### 3.1.2 Caractéristiques des objets locaux

**Sémantique.** Les classes et objets locaux ont le comportement suivant : les instances des classes locales sont liées à la machine virtuelle où elles ont été créées et ne peuvent pas être référencées par d'autres machines virtuelles de l'environnement.

**Création et accès.** La création et l'accès aux méthodes et attributs des objets locaux sont identiques à ceux de Java, valables à l'intérieur de la machine virtuelle où l'appel est effectué.

**Partie statique.** Une classe locale est chargée et initialisée séparément (à la demande) dans toute machine virtuelle qui participe à l'environnement distribué JavaParty. L'accès à des méthodes et attributs statiques concerne seulement la classe chargée dans la machine virtuelle courante.

**Passage des paramètres.** Les objets locaux peuvent être passés en paramètre pour une méthode distante et peuvent être retournés comme résultats, le passage étant réalisé par copie.

**Mobilité.** Les objets locaux ne peuvent pas migrés, leur localisation étant fixe dans la machine de leur instanciation. Si des objets locaux sont références par des objets remote, et ces derniers migrent, ils seront copiés dans la machine destinataire de la migration.

## 3.2 Exemple de programmation distribuée JavaParty

L'environnement de programmation JavaParty est conçu pour être le plus proche possible de la programmation Java. Seul le mot clé *remote* marque les classes remote et ainsi les objets remote.

```
remote class A {
   public int i;
   public static int si;
   public void mVoid (...) {...}
   public int mReturn (...) {  ... return ...;}
   public static void mStatique (...) {...}
}
```

Une classe remote en JavaParty est déclarée de la même façon qu'en Java, en ajoutant le mot clé *remote*. La déclaration d'une classe remote, comme dans l'exemple précédent peut contenir des attributs d'instance (*int i*), des attributs statiques (*static int si*), des méthodes d'instance (*mVoid, mReturn*) ou des méthode statiques (*mStatique*).

L'utilisation de cette classe n'apporte pas de modification par rapport à l'utilisation d'une classe Java classique :

```
a = new A();
a.i = 5;
A.si = 10;
a.mVoid(...);
int r = a.mReturn();
A.mStatique();
```

## 3.3 Mise en œuvre du modèle distribué

### 3.3.1 Classes et objets remote

Les classes et les objets remote en JavaParty sont introduits à l'aide du mot clé *remote*. Le code JavaParty n'est pas compatible avec le code Java, parce que le mot clé qui marque les objets remote n'appartient pas au langage Java. Un autre compilateur (voir la figure 3.1), fourni par JavaParty, génère du code compatible avec la machine virtuelle Java.

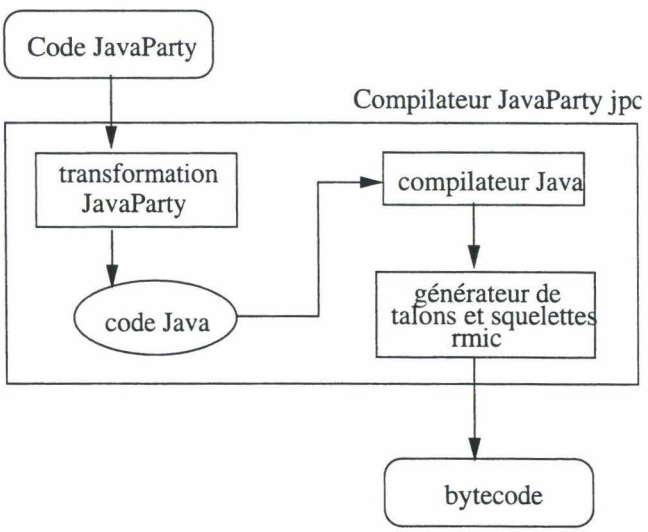



Figure 3.1 – Compilateur JavaParty

Le compilateur JavaParty génère trois classes et deux interfaces correspondant aux trois parties d'une classe remote :

- la classe proxy,
- l'interface et la classe pour la partie instance (définissant l'objet instance),
- l'interface et la classe pour la partie statique (définissant l'objet classe).

### Description générale

La classe proxy est la représentation locale de la classe remote et gère l'indirection d'un appel de méthode vers la partie instance ou statique, en fonction du type d'appel. Les parties instance et statique sont gérées par deux objets distants au sens Java RMI.

Le code généré par le compilateur apporte des modifications pour gérer la création à distance de la partie statique et de la partie instance d'une classe remote et l'accès à ces parties. Le compilateur JavaParty génère le code détaillé ci-dessous, pour la classe A, dont la définition est la suivante :

```
remote class A {
  public void m() {...}
  public static void mS() {...}
}
```

**La partie instance.** Toute méthode d'instance est réécrite dans la partie instance pour compter les appels : au début de la méthode, un compteur d'appels est incrémenté et, à la fin, le compteur est décrémenté. L'utilité du compteur est exploitée par le mécanisme de migration, détaillé dans la section 14.3.3.

```
public class A_instance_impl extends jp.lang.RemoteObject_instance_impl
                            implements A_instance_intf  {
  // la classe jp.lang.RemoteObject_instance_impl fait partie de la
  // bibliothèque de classes JavaParty et
  // l'interface A_instance_intf est générée par le compilateur de JavaParty
  ...
  public void m() {
    _inc();
    try {
        ...
    } finally { _dec(); }
  }
}
```

Des méthodes supplémentaires sont ajoutées pour donner accès à tout attribut d'instance.

**La partie statique.** Toute méthode statique est simplement recopiée dans la classe correspondante générée, en éliminant le modificateur *static*. La partie statique est gérée par un objet unique, dans tout l'environnement ; donc un accès statique est redirigé comme méthode d'instance, sur cet objet.

```
public class A_class_impl extends jp.lang.RemoteObject_class_impl
                             implements A_class_intf {
  // la classe jp.lang.RemoteObject_class_impl fait partie de la
  // bibliothèque de classes JavaParty et
```

```
  // l'interface A_class_intf est générée par le compilateur de JavaParty
  ...
  public void mS() { ... }
}
```

**La partie proxy.** La partie proxy est écrite de manière différente, pour le cas d'un appel statique ou d'instance : pour un appel statique, l'environnement d'exécution est utilisé pour accéder à l'objet statique de la classe, sur lequel la méthode sera invoquée et pour un appel d'instance, une référence interne, de type proxy, est utilisée pour accéder à la partie instance.

```
public class A extends jp.lang.RemoteObject {
  ...
  public void m() {
    ...
      // utiliser la référence interne ref pour invoquer l'appel distant
      ((A_instance_intf) ref).m();
    ...
  }

  public static void mS() {
    ...
      // récupérer l'objet classe gérant la partie statique de la classe A
      ((A_class_intf)jp.lang.RuntimeEnvironment.getClassObj("A")).mS();
    ...
  }
}
```

L'environnement d'exécution (présenté dans la section suivante) aide à retrouver l'objet classe qui gère la partie statique de la classe. En plus, des méthodes sont ajoutées pour l'accès aux attributs d'instance et statiques.

### L'héritage

La considération des classes remote dans une hiérarchie d'héritage apporte de nouvelles caractéristiques pour la génération de code (voir la figure 3.3).

La sémantique JavaParty prévoit qu'une classe peut être remote si elle n'hérite d'aucune autre classe, ou bien si elle hérite d'une autre classe remote. Ainsi, dans la déclaration suivante lors de la définition de la classe $A$, le mot clé marque la classe remote et, pour la classe $B$, en héritant de la classe $A$, le mot clé n'est plus nécessaire, $B$ étant considéré par défaut remote.

```
remote class A {...} // même que la définition précedente
class B extends A implements C {...}
```

Les figures 3.2 et 3.3 montrent un exemple de hiérarchie de classes générées pour la déclaration précédente de classes utilisateur. Les interfaces dans les figures sont représentées par des ellipses, et les classes par des rectangles. Le lien d'héritage entre les classes ou les interfaces est marqué en ligne continue, tandis que le lien d'implémentation des interfaces est en pointillé.

Lors d'une relation d'héritage entre deux classes remote, le compilateur génère les classes correspondantes qui conservent l'héritage pour la partie instance. L'héritage pour la partie instance

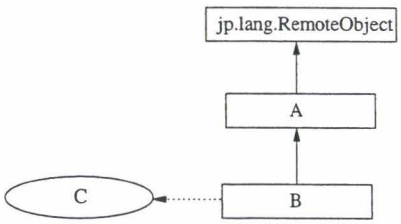



Figure 3.2 – Hiérarchie des classes proxy générées par le compilateur JavaParty

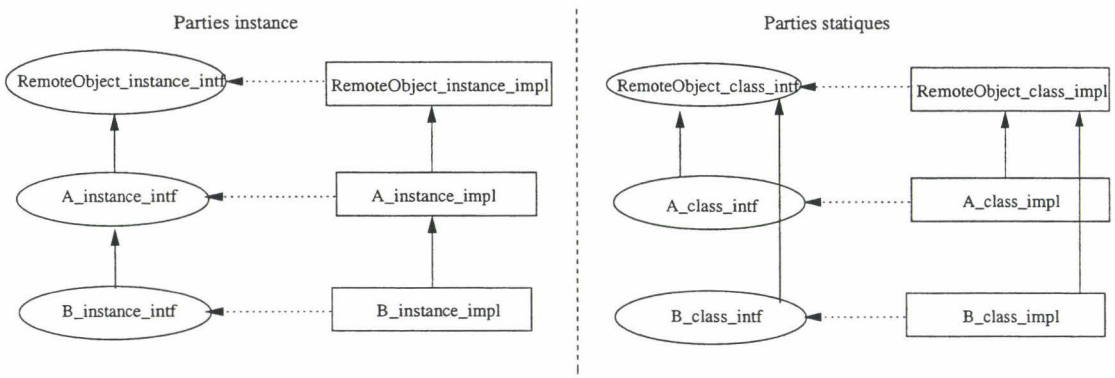



Figure 3.3 – Hiérarchie des classes, pour les parties instance et statique, générées par le compilateur JavaParty

(figure 3.3, à gauche) est conservé parce que les appels de méthodes d'instance d'une super-classe doivent être valides. Ainsi, pour une classe $B$ vide, mais qui hérite de la classe $A$, l'appel `new B().m()` doit être valide.

Le choix de ne pas conserver le schéma d'héritage pour la partie statique (figure 3.3, à droite) s'explique par le comportement de la machine virtuelle Java lors de l'initialisation d'une classe. En effet, la spécification de la machine virtuelle indique qu'avant l'initialisation d'une classe, toute super-classe sera initialisée, si elle ne l'a pas déjà été. Vu qu'un objet classe ne doit être initialisé qu'une seule fois dans tout l'environnement distribué, si un objet de type $B$ est utilisé dans une autre machine que l'objet classe de $A$, la spécification exige que la partie statique de $B$, et implicitement sa super-classe, donc la partie statique de $A$, soient initialisées, ce qui induit une double existence de l'objet classe de $A$, situation non acceptée dans la sémantique JavaParty. Ainsi, l'héritage est réalisé directement de la classe *RemoteObject_class_impl* et pas de la classe *A_class_impl*, pour l'objet classe de la classe $B$. Le même raisonnement explique l'héritage au niveau des interfaces pour la partie statique. L'interface $C$ intervient uniquement dans l'héritage des classes traitant la partie proxy, héritage qui est conservé par rapport à la déclaration initiale (voir la figure 3.2).

Toutes les classes générées par le compilateur de JavaParty sont utilisées de manière transparente par le programmeur.

### 3.3.2 Création d'objets remote

En Java, un objet est créé sur la machine virtuelle sur laquelle le *new* a été exécuté. Deux solutions existent pour instancier un objet à distance dans un environnement distribué :

- le constructeur est appelé sur la machine locale et l'objet ainsi créé est ensuite migré sur la machine destinataire,
- le constructeur est directement appelé sur la machine cible.

JavaParty a choisi la deuxième solution, à cause de la complexité de la migration, du nombre d'appels distants et du coût de la migration.

Comme on l'a vu précédemment, il y a un seul objet classe (par classe remote) instancié sur une machine virtuelle dans tout le système distribué. Donc, c'est seulement sur cette machine que l'on pourrait créer, à travers les constructeurs, les objets remote. Pour avoir la possibilité d'instancier des objets sur n'importe quelle machine virtuelle, on a des "fausses" copies de la partie statique, sur chaque machine, qui serviront à l'instanciation d'un objet remote.

Les deux étapes liées à l'instanciation d'un objet remote sont présentées dans la figure 3.4.

Premièrement, le constructeur de la classe correspondante est récupéré, sous forme d'objet et instancié (ainsi est construite et initialisée la partie statique de la classe) et deuxièmement, le constructeur correspondant à la partie instance est appelé (pour l'initialisation des membres d'instance de l'objet). Le proxy a la fonctionnalité de déclencher cet enchaînement d'initialisations et en général, pour tout appel, de le diriger vers l'objet traitant la partie instance ou statique d'une classe.

La récupération du constructeur d'une classe remote est réalisée par un appel à l'environnement local d'exécution (*RuntimeEnvironment*, voir la section 3.4). Si la référence n'est pas présente, ce qui correspond au cas d'une première initialisation, la référence est cherchée auprès du composant central (*RuntimeManager*, voir la section 3.4) et enregistrée localement afin que tout accès ultérieur soit local.

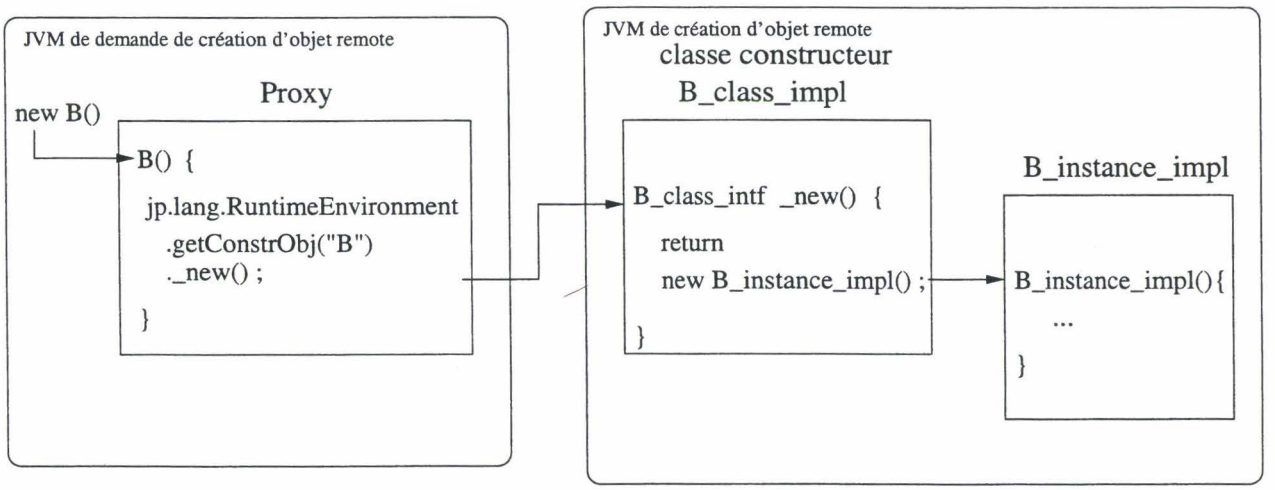



Figure 3.4 – Le schéma d'instanciation d'un objet remote

### 3.3.3 Migration d'objets remote

La migration en JavaParty est réalisée au niveau d'un objet remote. De plus, il s'agit d'une migration de données uniquement. Une contrainte importante est imposée pour migrer avec succès un objet remote. Un objet remote ne peut pas migrer tant qu'une méthode est en cours d'exécution. Une méthode est considérée en cours d'exécution si son code a commencé à être exécuté par le

processeur, mais n'a pas encore été fini. Cette contrainte est due à l'impossibilité d'accès aux informations liées à la pile d'exécution d'une méthode dans la machine virtuelle Java.

Les étapes de la migration en JavaParty, présentées en détail dans la section 14.3.3, se résument :
- à la construction de l'état des données,
- au transfert de l'état sur la machine destinataire.

Tenant compte que le langage Java permet le transfert de bytecode, JavaParty ajoute à la migration, seulement, le transfert des données. La construction de l'état des données est réalisée par la technique de sérialisation Java.

Le contrôle de la migration permet d'acheminer les invocations de méthodes sur un objet, arrivant pendant sa migration, par le mécanisme de redirection. Ainsi, un objet qui migre laisse une trace de son déplacement par l'intermédiaire d'un proxy, et les méthodes arrivant après la migration de l'objet seront redirigées, grâce au proxy, vers la nouvelle localisation de l'objet. Les méthodes arrivant pendant la migration elle-même ne sont pas exécutées, elles sont gardées chez l'appelant jusqu'à ce que le proxy de l'objet soit mis à jour.

## 3.4 L'environnement d'exécution

L'environnement d'exécution JavaParty est composé de plusieurs JVM qui sont accueillies dans une même ou dans différentes machines physiques (stations). Les machines physiques utilisées sont connectées en réseau.

L'environnement d'exécution est formé par un composant central (*RuntimeManager*) et des machines virtuelles appelées *locales* (*VirtualMachine*), qui s'enregistrent auprès du composant central. L'application distribuée est lancée dans une nouvelle JVM et la distribution des objets remote est décidée par le composant central, en fonction de la politique de placement.

La figure 3.5 montre le schéma des dépendances de différents composants de l'environnement distribué d'exécution, leur description est détaillée ci-dessous.

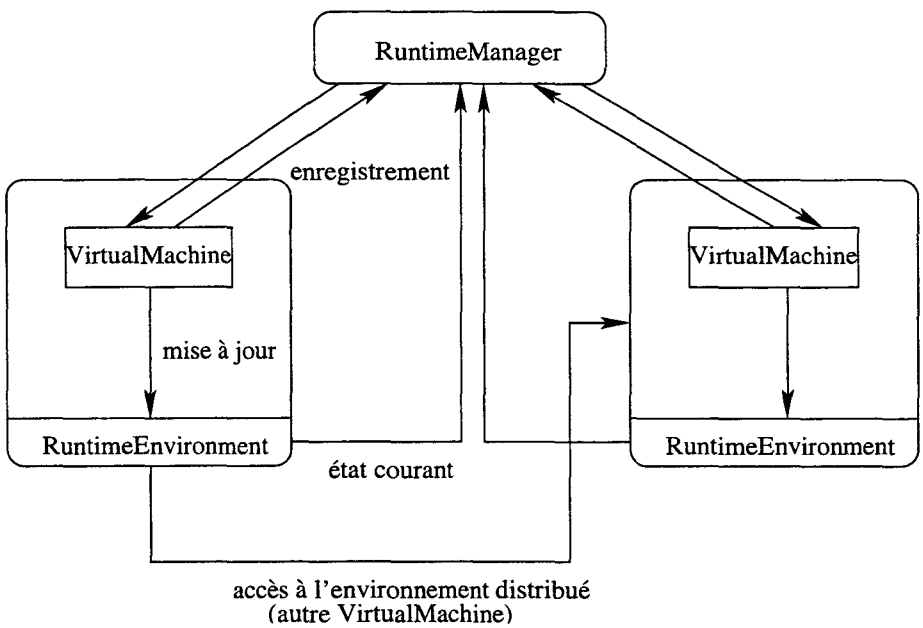



Figure 3.5 – Le schéma de l'environnement d'exécution JavaParty

### 3.4.1 Description des composants

**RuntimeManager.** Le RuntimeManager est le composant central d'un environnement distribué en JavaParty. De manière interne, il détient :

- une liste des machines virtuelles appartenant à l'environnement distribué, accessibles à distance,
- une table des objets classes (utilisés pour l'initialisation de la partie statique d'une classe),
- une table des objets constructeur (utilisés pour l'initialisation de la partie instance d'un objet remote).

Ce composant central démarre et arrête l'environnement distribué et gère également le lancement de l'application dans l'environnement. Il enregistre les nouvelles machines virtuelles et leur offre des services de type envoi d'un objet constructeur ou d'un objet classe. Si une nouvelle classe est chargée dans l'environnement distribué pour la première fois, l'objet classe correspondant n'est pas disponible sur la machine d'instanciation, ni auprès du composant central. Dans ce cas, le *RuntimeManager* se charge de la création d'un nouvel objet classe.

**VirtualMachine.** La machine virtuelle *VirtualMachine* offre les services suivants :

- création d'un nouvel objet constructeur pour une classe remote sur la machine virtuelle locale,
- création d'un nouvel objet classe pour une classe remote sur la machine virtuelle locale,
- initialisation de l'environnement local d'exécution (présenté dans la suite).

Lors de la migration d'un objet remote, la machine virtuelle gère la création de la nouvelle instance (à travers le mécanisme de désérialisation - voir la section 3.3.3).

Les machines virtuelles sont identifiées par des entiers, commençant à zéro.

**RuntimeEnvironment.** L'environnement d'exécution local *RuntimeEnvironment* permet l'accès aux objets remote et locaux dans l'environnement distribué, ayant des références vers la machine virtuelle locale et le composant central (voir la figure 3.5). Essentiellement, le *RuntimeEnvironment* offre les services de :

- récupération d'un objet classe particulier (en passant éventuellement par le composant central s'il n'existe pas localement),
- récupération d'un objet constructeur particulier (passage possible par le composant central).

Toutes les méthodes offertes par l'environnement local d'exécution sont statiques, accessibles par le biais d'un appel local, dans le cadre de la machine virtuelle.

### 3.4.2 Déploiement de l'application

Une application distribuée JavaParty s'exécute dans l'environnement présenté auparavant, composé de plusieurs machines virtuelles. Le déploiement de l'application dépend de la politique de placement des objets remote en JavaParty à l'instanciation, et des demandes explicites de migration au cours de l'exécution de l'application.

A l'instanciation, le placement des objets remote peut être décidé de trois manières différentes :

- Si le programmeur décide de ne pas se soucier du problème de placement des objets, l'objet est placé de manière aléatoire, sur une des machines virtuelles de l'environnement d'exécution.
- Des classes de distribution peuvent être développées et installées dans l'environnement d'exécution, pour orienter le placement des objets remote et des objets classes. De cette manière le programmeur interagit avec l'environnement, sans changer son application. Ce type de distribution est un paramétrage de l'environnement d'exécution, explicitement donné par le programmeur.

- Les décisions d'allocation peuvent être influencées directement par des appels de l'API JavaParty dans le code de l'application. Dans ce cas, chaque instanciation d'un objet remote est accompagnée d'une indication sur la localisation voulue pour l'objet (sous la forme de numéro de JVM). L'inconvénient de cette approche est le mélange entre le code du programmeur et le code de distribution. Cette manière manque de flexibilité par rapport aux caractéristiques de l'environnement (ressources, etc.).

Le placement, à l'instanciation, des objets remote peut être changé pendant l'exécution de l'application par des directives de migration des objets d'une machine virtuelle à une autre.

# Deuxième partie

# Méthodologie de programmation des applications distribuées et parallèles

# Avant propos

Le contexte retenu est basé sur un modèle d'objets JavaParty avec les caractéristiques de création à distance transparente et de migration d'objets, avec un protocole de communication basé sur RMI, et un modèle d'exécution formé de machines virtuelles Java accueillies par des stations d'une grappe.

Cette partie se focalise sur la proposition d'une méthodologie pour la conception aisée d'application Java distribuées et parallèles.

L'expression d'un parallélisme sous ses différentes formes (chapitre 4) se retrouve dans plusieurs projets (chapitre 5). La proposition d'une expression facile du parallélisme, ainsi que la récupération complètement asynchrone des résultats sont présentées dans le chapitre 6. Ces outils sont réunis dans une bibliothèque d'outils parallèles ADAJ, dont les aspects techniques sont décrits dans le chapitre 7. Le chapitre 8 met en évidence la méthodologie de programmation ADAJ à travers différentes applications. Finalement, le chapitre 9 évalue le coût d'utilisation des outils parallèles ADAJ en comparaison avec une utilisation JavaParty.

# Chapitre 4

# Introduction

La conception d'une application parallèle et distribuée exige de s'appuyer sur une méthodologie et des outils qui facilitent ce travail. Ces mécanismes peuvent être basés sur des modèles de conception, des bibliothèques ou des frameworks. Le choix de cet environnement de programmation n'est pas simple et, de plus, deux types de problèmes se posent : comment exprimer le parallélisme, sous quelle forme, et comment réaliser la distribution des applications sur un ensemble de stations.

Les différentes caractéristiques liées à l'expression du parallélisme, aux types d'outils d'aide à la conception ou aux aspects de la distribution, ainsi que l'approche Java par rapport à toutes ces notions, seront présentées ci-dessous.

## 4.1 Outils de l'aide à la conception

L'utilisation d'un modèle de programmation peut être maîtrisée par des outils génériques et flexibles permettant au développeur d'écrire son programme parallèle et distribué de manière simple et intuitive. Ces outils appartiennent à des modèles de conception, frameworks ou bibliothèques, approches différentes ayant pour objectif l'aide à la conception des applications parallèles.

**Modèle de conception** Les modèles de conception permettent de décrire des solutions récurrentes pour des problèmes standards. En utilisant les modèles de conception, le développeur se base sur des solutions déjà existantes, plutôt que les réinventer.

**Framework** Un framework est constitué d'un ensemble de composants intégrés qui collaborent pour fournir une architecture réutilisable. Un framework offre un cadre de programmation dans lequel l'utilisateur est contraint à redéfinir certaines fonctionnalités pour pouvoir profiter des services offerts. Par rapport à un modèle de conception, qui est généralement conçu indépendamment d'un langage de programmation, un framework est plus couramment implémenté dans un langage concret. Les deux outils d'aide à la programmation, le modèle de conception et le framework, se rejoignent dans les techniques d'assistance à la conception des applications.

**Bibliothèque** Les bibliothèques offrent des fonctionnalités d'ordre général pour l'implémentation de nombreuses fonctions utilisées fréquemment. Une bibliothèque est conçue de telle sorte que les détails d'implémentation soient cachés à l'utilisateur. L'usage d'une telle bibliothèque n'est pas obligatoire pour exprimer certaines fonctionnalités.

## 4.2 Programmation parallèle

Le modèle de programmation parallèle est sémantiquement plus riche qu'un modèle purement séquentiel, grâce à l'expression du parallélisme sous ses différentes formes : la concurrence, le parallélisme de contrôle, de données et de flux.

Le modèle parallèle suppose l'existence d'actions qui peuvent être exécutées en parallèle, et de constructions permettant leur expression. Par rapport à un modèle séquentiel qui contient un seul flot de contrôle, s'exécutant à un instant donné, le modèle parallèle se caractérise par l'existence de plusieurs flots de contrôle, correspondant aux différentes actions parallèles. Les multiples flots de contrôle apparaissant dans un modèle parallèle sont généralement exécutés à l'intérieur d'une même entité. Celle-ci, appelée *processus*, est donc constituée de plusieurs tâches, sous-unités de calcul d'un processus, aussi appelées *threads* (ou *processus légers*). La concurrence de plusieurs threads est acquise grâce à la technique du *multithreading*.

L'utilisation du multithreading, au lieu de plusieurs processus monothreadés, résulte des motivations suivantes :

- le surcoût engendré par la création d'un nouveau processus est plus important que celui de la création d'un thread dans un processus,
- le changement de contexte entre les threads d'un même espace d'adressage est moins coûteux que le changement de contexte entre des processus qui ont leur propre espace d'adressage,
- les threads permettent le parallélisme, combiné avec une exécution séquentielle et des appels système bloquants,
- le partage des ressources peut être réalisé, plus facilement, entre les threads d'un processus qu'entre les processus, parce que les threads d'un même processus partagent le même espace d'adressage.

Une complexité supplémentaire apparaît lorsque le modèle parallèle nécessite une coordination et des synchronisations entre les différents flots de contrôle (qu'ils soient des threads ou des processus).

### 4.2.1 Formes de parallélisme

La richesse d'un modèle parallèle se manifeste dans la variété des différentes approches complémentaires de l'expression de ce parallélisme. L'expressivité, c'est-à-dire la possibilité de fournir un supplément d'information, engendrée par l'utilisation simultanée de plusieurs formes de parallélisme, est censée offrir, en même temps, une meilleure efficacité lors de l'exécution. Le compromis entre l'expressivité de l'environnement et l'utilisation efficace des systèmes d'exécution est un enjeu important.

#### La concurrence

La concurrence résulte de la simultanéité de tâches qui se partagent le temps d'utilisation d'une ressource. Ceci engendre un type de parallélisme appelé *parallélisme virtuel*. Cette concurrence n'engage pas forcément un vrai parallélisme, parce qu'à un instant donné, un seul processus s'exécute véritablement. Sur des systèmes à temps partagé, les processus ont des tranches de temps pour s'exécuter (fixe ou variable en fonction de la priorité), ce qui donne l'impression générale que plusieurs tâches sont globalement en train de s'exécuter. Un coût supplémentaire apparaît : celui lié au changement de contexte des processus.

Des mécanismes de synchronisation doivent être prévus pour une exécution concurrente correcte.

### Le parallélisme de contrôle

Le parallélisme de contrôle (de tâches) se base sur le principe d'exécution des flots de contrôle en parallèle. La difficulté dans le parallélisme de tâches est l'obligation pour le programmeur d'extraire des tâches indépendantes et, éventuellement, de les affecter à des processeurs distincts. Le découpage en tâches indépendantes n'élimine pas la possibilité de communication entre des tâches différentes ou leur accès concurrent sur les mêmes données. Ainsi, ce type de parallélisme suppose la spécification de leurs interactions, par des mécanismes de communication ou de synchronisation.

### Le parallélisme de données

Dans un calcul exploitant le parallélisme de données, la même opération peut être exécutée, simultanément, sur des données différentes. Le modèle de parallélisme de données comporte deux avantages : il facilite la programmation - une seule définition de traitement - , et il est facilement extensible - les programmes correspondant aux problèmes de plus grande taille sont facilement étendus. Le modèle correspond bien aux situations où il est possible de diviser les données en sous-ensembles sur lesquels des copies d'un même programme s'exécutent. L'indépendance des données offre un caractère complètement parallèle aux applications (appelées *embarassingly parallel applications*). Toutefois, il est souvent nécessaire de prévoir des phases d'échange d'informations, et, ainsi, des synchronisations périodiques, après des phases de calcul parallèle. Un exemple de ce type de modèle parallèle de programmation est le modèle BSP (Bulk Synchronous Programming) [WA99].

### Le parallélisme de flux

Le parallélisme de flux est exprimé par une séquence de tâches appliquées à des séries de données similaires. Les tâches, constituant les différentes étapes du processus, sont indépendantes grâce au mécanisme de *pipeline*. Un pipeline est constitué d'étages, chacun étant dédié à un traitement particulier. Les étages du pipeline travaillent de façon concurrente sur des données différentes. Le nombre d'étages du pipeline dénombre la quantité maximale de tâches qui peuvent être réalisées en parallèle.

## 4.2.2 Gestion du degré et granularité du parallélisme

Le parallélisme est caractérisé par deux attributs : la granularité (l'importance des tâches exécutées en parallèle, c'est-à-dire leur temps d'exécution) et le degré (combien de tâches s'exécutent en parallèle).

Le parallélisme exprimé dans les programmes se décline sur trois niveaux de granularité, selon le parallélisme. Il s'agit du niveau :

- des routines - *granularité grossière*,
- d'un ensemble d'instructions - *granularité moyenne*,
- des boucles - *granularité fine*.

Le parallélisme à gros grain engendre, en général, un nombre faible de tâches s'exécutant en parallèle. En conséquence, le nombre d'activations de tâches est réduit, d'où une diminution du nombre de changements de contexte entre les processus. En même temps, le traitement séquentiel étant important, d'éventuelles communications, en nombre réduit, sont nécessaires. Elles sont proportionnellement moins importantes que les calculs (Cette situation est caractérisée par le fait que les communications sont recouvertes par les calculs). Mais, lorsque les tâches deviennent inactives, par exemple, en attente de résultats, le processeur peut se trouver dans un état d'inactivité.

Dans le parallélisme à grain fin, le traitement à exécuter est divisé en un nombre important de tâches de petite taille. Contrairement au cas précédent, ce type de granularité facilite la recherche d'une autre tâche à exécuter en cas d'inactivité de certaines tâches, grâce au principe de la concurrence. Par contre, le nombre de changements de contexte liés aux activations des tâches est élevé et les communications éventuelles apparaissant entre les tâches peuvent être importantes.

La granularité dépend du temps du traitement parallèle, qui est parfois lié à la quantité de données à traiter. Dans aucun des modèles de programmation parallèle, la granularité n'est gérée de manière automatique. Implicitement, la granularité est définie par les traitements que spécifie le développeur.

Le degré de parallélisme fait référence aux opérations pouvant s'exécuter en parallèle. En pratique, le degré du parallélisme est le nombre de tâches s'exécutant en parallèle. En particulier, pour le modèle de programmation à parallélisme de données, le degré est le nombre de sous-ensembles fragmentés sur lesquels les tâches s'exécutent. Pour le modèle du parallélisme de flux, le degré est le nombre d'étages du pipeline.

### 4.2.3 Transparence et facilité d'expression du parallélisme

L'expression du parallélisme des applications, sous les différentes formes présentées auparavant, introduit deux aspects : la transparence, au niveau conceptuel, et la facilité, au niveau de l'écriture.

La transparence se décline sous deux types :
- une transparence fonctionnelle,
- une transparence syntaxique.

La transparence désigne une propriété qualitative de l'expression du parallélisme. La transparence fonctionnelle résulte de la gestion cachée du parallélisme et de l'introduction de mécanismes de synchronisation. La transparence syntaxique, concernant l'écriture, prend deux formes : totale, qui coïncide avec une écriture classique des appels et partielle, qui est générée par la volonté du développeur d'une expression parallèle, de faire la différence entre un appel classique et un appel parallèle.

La transparence conceptuelle de l'expression du parallélisme se reflète dans la facilité d'écriture. Rendre l'expression du parallélisme simple et intuitive est l'un des objectifs des concepteurs d'outils d'aide à la conception.

### 4.2.4 Contrôle du parallélisme

Le contrôle du parallélisme est réalisé à deux niveaux : programmation, par des mécanismes de synchronisation et communication entre les processus (détaillés par la suite) et, au niveau ordonnancement, par des politiques d'allocation des processeurs aux différents processus. Le mécanisme d'ordonnancement concerne moins l'aspect conception, raison pour laquelle il sera détaillé dans la troisième partie de cette thèse, liée à l'efficacité de l'exécution.

**Synchronisation** Les tâches d'un même processus partagent le même espace d'adressage. Des mécanismes de synchronisation sont intégrés pour gérer l'accès simultané de plusieurs tâches à une même donnée.

De façon classique, les synchronisations entre les tâches concernent le problème de l'exclusion mutuelle, qui assure l'accès unique à une ressource partagée. Ainsi, des actions non-compatibles effectuées par deux tâches sont groupées dans une *section critique*. Les techniques utilisées pour résoudre le problème d'exclusion mutuelle sont les *variables d'exclusion mutuelle* et *de condition*.

**Ordonnancement** Du point de vue du développeur, la conception d'un système multitâches est réalisée de manière indépendante de l'architecture de la plate-forme matérielle sur laquelle l'application s'exécute. Lors de l'exécution de plusieurs tâches, le comportement est contrôlé par *l'ordonnanceur* (*scheduler*), par un procédé capable de gérer plusieurs tâches simultanément.

## 4.3 Distribution des programmes parallèles

Exprimer du parallélisme sur un système monoprocesseur n'apporte pas d'efficacité[1], le parallélisme étant sous sa forme virtuelle. La distribution sur un système multiprocesseur s'impose si de meilleures performances sont souhaitées. La distribution concerne certaines entités, qui auront la propriété d'accessibilité à distance. Cette propriété implique la possibilité de partage de l'entité par plusieurs tâches. Rendre des entités distantes impose des changements dans les caractéristiques par rapport aux entités locales. Ces caractéristiques comportent la sémantique, l'accès et le placement des entités, aspects détaillés par la suite.

### 4.3.1 Déploiement des applications

La distribution d'une application consiste à répartir ses composants sur les machines de la grappe. Pour les applications orientées objets, les composants de base concernés sont les objets. Ainsi, l'objet est l'entité distribuable d'une application. Les questions soulevées par la distribution des objets concernent :

- la modalité de distribution, qui suppose d'indiquer les objets distribuables,
- l'accès aux objets distribuables, et notamment comment est réalisée la communication entre deux objets distants,
- le placement des objets distribués, par rapport à différentes politiques.

#### Marquage des objets

Lors de la considération d'une application à répartir, certains objets sont marqués comme distribués. Le marquage peut être réalisé à deux niveaux :

- celui de la déclaration de la classe (marquage répercuté à toute instance de la classe),
- celui d'une instance particulière, par des directives de marquage ou l'emploi de méthodes de bibliothèque.

Le marquage au niveau de la classe offre une déclaration statique d'objets distribués : un objet d'une classe marquée sera toujours distribué. Le marquage d'une instance particulière reflète une dynamicité par la possibilité de rendre un objet distribué, lors de l'instanciation ou après. Le problème soulevé dans ce cas est lié à la sémantique différente de l'objet et à la cohérence qui doit être assurée.

#### Communication entre les objets

Le modèle classique de communication entre les objets repose sur l'invocation de méthodes. Etant donné que la communication entre les objets distribués passe au travers du réseau, une invocation de méthode peut être transparente ou non-transparente. La transparence assure un même type de communication entre objets distribués comme entre objets locaux, tandis que la non-transparence n'offre pas cette caractéristique.

[1] sauf dans le cas de processus multithreadés

**Placement des objets**

Le programmeur d'applications parallèles et distribuées n'est pas concerné par la localisation des objets si l'environnement de programmation a été conçu transparent. Toutefois, la distribution des objets a des implications sur la performance de l'exécution d'une application. Une gestion de données judicieuse est reflétée par des critères de placement induits par les contraintes suivantes :

- minimiser les connexions en groupant les opérations,
- optimiser le nombre d'opérations réalisées,
- minimiser les mises à jour.

Ces contraintes peuvent être respectées par deux types de distribution, en fonction du niveau de l'intervention des directives :

- dans le code : *distribution explicite*,
- dans le middleware : *distribution implicite*,
  - distribution par défaut,
  - distribution automatique.

La distribution explicite des objets réalisée au niveau du code source profite de la connaissance qu'a le programmeur du comportement de l'application. Ce type de distribution est géré en utilisant soit des primitives de placement offertes au travers d'une bibliothèque, soit des directives du programmeur qui indiquent des placements convenables pour l'application.

La distribution implicite propose des politiques de placement par défaut du type cyclique ou par bloc, indépendamment du support d'exécution. Par extension, les caractéristiques de la plate-forme support peuvent être prises en compte et des directives implicites génériques peuvent être spécifiées.

La distribution automatique suppose une transparence totale, pour l'utilisateur, du placement des objets. Cette approche est basée sur l'extraction automatique du parallélisme, et en déduit, par une analyse du code, une distribution optimale des objets.

### 4.3.2 Modèles d'objets distribués

La conception des applications distribuées dans un contexte orienté objets est facilitée par la définition et la mise en œuvre d'un modèle d'objets intégrant leur distribution sur un ensemble de stations d'un réseau. Ces objets sont interconnectés par le biais de communications à travers le réseau.

Les technologies qui offrent un support pour les architectures d'objets distribués sont : RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture) et DCOM (Distributed Component Object Model). Les trois types d'architectures s'appuient sur le modèle client-serveur qui structure une application en entités demandant des services (*clients*) et en entités qui offrent des services (*serveurs*). Les requêtes exprimées par les clients pour obtenir un service sont traitées par le serveur qui leur renvoie le résultat. Plusieurs requêtes peuvent arriver simultanément au serveur, et nécessitent, de la part du serveur, la gestion d'une concurrence.

**RMI,** intégré dans le langage à partir de la version 1.1, est l'implémentation de JavaSoft du modèle d'objets Java distribués. RMI permet la description des applications clients et serveurs qui communiquent par des invocations de méthodes sur les objets. Le protocole natif utilisé, JRMP (Java Remote Method Protocol), permet la communication uniquement entre des objets Java RMI (la description détaillée est réalisée dans le chapitre 2).

**CORBA,** développé par OMG (Object Management Group [OMG]), permet des invocations de méthodes sur des objets résidant sur le réseau, indépendamment de leur placement, tout comme en local. CORBA apporte, entre les modèles distribués, l'interopérabilité entre des objets, grâce au protocole IIOP (Internet Inter-ORB Protocol), qui permet de faire communiquer des objets indépendamment de leur plate-forme d'accueil ou du langage dans lequel ils sont écrits.

**DCOM** est la solution Microsoft pour des architectures à objets distribués. DCOM [FR97] se base sur le modèle COM (Component Object Model) qui fournit un ensemble d'interfaces permettant aux clients de communiquer à l'intérieur d'un même ordinateur. DCOM étend ce modèle pour supporter la distribution des objets sur le réseau.

## 4.4 Programmation parallèle et distribuée en Java

Le langage à objets Java est fondé sur le concept d'objet, qui encapsule des données et du code. L'entité (la ressource) est modélisée par un objet et le traitement (processus ou tâche) par une méthode. L'expression du parallélisme en Java, la projection des problèmes soulevés pour la gestion du parallélisme et de la distribution sont présentées ci-dessous.

### 4.4.1 Modèle de programmation parallèle en Java

Le parallélisme abstrait comprend deux formes implémentées : le *parallélisme virtuel* (*pseudo-parallélisme*), pour des tâches s'exécutant en tranches de temps allouées par un processeur, et le *parallélisme physique* dans le cas d'exécution de plusieurs tâches en même temps sur plusieurs processeurs.

Le parallélisme en Java s'exprime à l'aide des *threads*. Un thread Java est un contexte d'exécution (processus léger) qui constitue un flux séquentiel de contrôle dans un programme. Tous les threads (d'un processus) existent dans un même espace d'adressage, partageant les variables globales mais un thread détient sa propre pile et son compteur d'instructions, ayant des ressources propres.

Avant la version 1.2, le parallélisme était exprimé sous la forme virtuelle. Sur une seule machine virtuelle Java, plusieurs threads offrent l'impression d'exécution simultanée mais en réalité, ils sont entrelacés sur le même processeur et un seul thread est exécuté à la fois. La JVM exécute des instructions pour un thread, puis des instructions pour un autre.

A partir du JDK1.2, sous le système d'exploitation Solaris, l'utilisation des threads natifs permet l'exploitation de plusieurs processeurs par un seul programme Java. Mais, étant dépendant du système d'exploitation utilisé, cet usage ne conserve plus la portabilité du Java.

La concurrence en Java est permise grâce à la technique du multithreading. En Java, un programme s'exécute dans un processus, celui de la machine virtuelle Java. Le parallélisme dans cette machine peut être réalisée uniquement grâce aux threads internes à la JVM. Java inclut des primitives de multithreading dans le langage lui-même, exprimant un parallélisme de contrôle. Les autres modèles de programmation parallèle (parallélisme de données ou de flux) ne sont pas présents a priori dans le langage Java.

### 4.4.2 Transparence et facilité d'expression du parallélisme en Java

L'expression du parallélisme en Java est totalement explicite par l'utilisation des threads. Lors de l'expression du parallélisme dans une application Java, le concepteur est amené à :

- définir une classe qui modélise le thread (en étendant la classe *java.lang.Thread* ou en implémentant l'interface *java.lang.Runnable*),
- définir le code à exécuter dans une méthode spécifique (en surchargeant la méthode *public void run ()*),
- créer le thread et le lancer (à l'aide de la méthode *start()* de la classe *Thread*).

Java impose la déclaration d'un comportement parallèle unique par classe du type thread. Ainsi, le code à exécuter est défini dans une méthode de signature fixe qui n'accepte pas de paramètres et ne renvoie pas de résultats.

Cette contrainte impose que le passage des paramètres, nécessaires éventuellement pour la méthode, soit fait à la création du thread ou par une méthode d'initialisation. Le même problème se pose pour le retour éventuel des résultats. La définition d'un thread en héritant de la classe de bibliothèque Java consomme l'héritage de classes ; l'usage de l'interface *Runnable* permet de conserver la hiérarchie des classes du programmeur.

### 4.4.3 Contrôle des threads en Java

**Synchronisation** En Java, tout objet possède un verrou d'exclusion mutuelle, manipulé à travers le mot clé *synchronized*. Le verrouillage peut être réalisé au niveau :

- d'un objet,
- d'une méthode (d'instance ou statique).

En Java, les variables de condition sont présentes en collaboration avec le mécanisme de notification. Le contrôle des accès à une ressource partagée, par notification, est réalisé grâce aux primitives *wait* et *notify/notifyAll* attachées à tout objet. Elles interviennent seulement pour les accès synchronisés des threads sur ces objets.

**Ordonnancement** Le principe d'ordonnancement mis en œuvre par la JVM est que le thread actif courant doit être le thread de plus haute priorité parmi tous les threads prêts.

Un contrôle explicite de l'exécution des threads Java est réalisé grâce à un certain nombre de primitives[2] : détruire le thread courant (*destroy*), interrompre le déroulement du thread courant (*interrupt*), positionner une barrière en attente de la fin du thread courant (*join*), endormir le thread courant pendant un certain temps (*sleep*), rendre la main à l'ordonnanceur (*yield*).

### 4.4.4 Distribution des programmes en Java

L'architecture d'objets distribués en Java est RMI, basée sur le modèle client-serveur, qui permet l'interaction uniquement entre des objets Java. Ces objets, instanciés sur des machines virtuelles différentes, sont appelés *objets distants*.

Le marquage des objets distants, présenté dans le modèle de programmation de la section 2.3, est introduit au niveau de la déclaration des classes distantes, au travers de l'héritage des interfaces ou classes spécifiques.

La communication entre les objets Java, par le biais d'invocation des méthodes, reste explicite, en traitant une exception spécifique, due à l'accès à travers le réseau. Ce type de communication, synchrone, est considéré comme non-transparent pour l'utilisateur, qui traite différemment les appels sur des objets locaux et les appels sur des objets distants. Le placement des objets est explicite, grâce au lancement, réalisé par l'utilisateur, de l'application serveur.

[2] dont quelques unes obsolètes à partir de la version 1.2 du langage

# Chapitre 5

# Etat de l'art

Dans ce chapitre, différents projets sont présentés mettant en évidence les caractéristiques des outils d'aide à la conception des applications parallèles et au déploiement des applications distribuées. Trois types de projets sont présentés : des environnements de programmation et d'exécution dédiés aux applications C/C++ (Dome, ACADA), des bibliothèques parallèles Java (JGL, DPJ et Ajents) et des environnements de développement et d'exécution d'applications Java (Jacob, ProActive, Do!).

## 5.1 Dome (Distributed Object Migration Environment)

Dome [ABL+95], développé à l'Université Carnegie Mellon, est conçu comme un environnement qui rend aux programmeurs une interface simple et intuitive pour la programmation parallèle. Dome est construit comme une bibliothèque de classes C++ qui utilise PVM[1] [WA99] pour le contrôle des processus et pour la communication.

Dome utilise le modèle SPMD (Single Program Multiple Data) pour paralléliser explicitement les programmes. Le parallélisme est exprimé par des exécutions, en parallèle, des copies du même programme sur des sous-ensembles de données de chaque objet Dome. Le nombre de copies, d'un même programme, est un par nœud[2], nombre qui peut être également contrôlé par l'utilisateur au démarrage de l'environnement Dome.

Une classe Dome représente en général une collection importante de données similaires comme, par exemple, un vecteur. Quand un objet de ce type de classe est instancié, il est automatiquement partitionné et distribué entre tous les processus du programme distribué. Les distributions proposées sont données par des directives :

- directive *whole* - tous les éléments sont dupliqués dans tous les processus,
- directive *block* - les éléments de l'objet Dome sont divisés équitablement entre les processus dans des blocs contiguës,
- directive *dynamic* - indique une distribution similaire à celle par blocs mais les données seront redistribuées dynamiquement entre les processus par le mécanisme d'equilibrage de charge activé aux intervalles de temps donnés.

Grâce au langage C et à sa facilité de surcharge d'opérateurs, les calculs, en utilisant ces opérateurs, sont réalisés en parallèle sur les nœuds de la machine virtuelle PVM. La surcharge offre au programmeur une simple manipulation des objets Dome et cache des détails de parallélisme.

[1] Parallel Virtual Machine
[2] composant d'une PVM

Par exemple, la multiplication de deux vecteurs distribués sur des nœuds de l'environnement s'exprime de manière transparente :

```
prod = vector1 * vector2;
```

La distribution initiale en Dome ne tient pas compte des différentes charges des nœuds et Dome fait la supposition que les mêmes calculs sur des données différentes ont la même granularité.

Dome n'offre pas d'expression d'un parallélisme de tâches, aspect compensé par un mécanisme d'équilibrage de charge qui profite des temps de synchronisation donnés par le modèle SPMD de programmation.

## 5.2 ACADA (Aide à la Conception d'Applications Distribuées Adaptatives)

Acada [VDL98], développé au LIFL[3], en France, est un environnement de programmation qui vise à simplifier la tâche du programmeur dans le développement des applications irrégulières et dynamiques orientées objets.

L'approche programmation proposée en Acada se base sur l'utilisation des données fragmentées et distribuées. La fragmentation des données est réalisée grâce aux annotations du programmeur qui précise la localisation qu'il souhaite réaliser pour certaines données.

La fragmentation initialement appliquée peut être remise en cause pendant l'exécution par une redistribution (*refragmentation*).

Les opérateurs parallèles attachés aux entités Acada sont les suivants :

- *spawn* - qui correspond à un appel asynchrone de fonction,
- *distribute* - qui correspond à une multiple exécution de spawn sur plusieurs données,
- *pipe* - qui correspond à une succession d'invocations des fonctions sur plusieurs données.

Ainsi, lors de l'exécution des applications Acada, le parallélisme de données s'associe au parallélisme de tâches.

La gestion du degré du parallélisme est dynamique, par la migration des objets ou par la refragmentation. La granularité du parallélisme est fixée par le programmeur. Ainsi, il est possible de définir la granularité d'un objet en fonction du nombre de processeurs, des spécificités des machines ou en fonction du contenu de l'objet. Acada oriente la programmation parallèle vers un grain fin du parallélisme.

Développé sur l'environnement multithreadé $PM^2$ [NM95], Acada traite les applications C/C++ en profitant du mécanisme de passage des fonctions comme paramètres. La refragmentation prévue risque d'être coûteuse dans la phase d'exécution de l'application, ainsi des points explicites de refragmentation sont placés dans le programme.

## 5.3 JGL (Java Generic Library)

JGL [Obj96], développé par Recursion Software, Dallas, est une bibliothèque 100 % Java qui a été conçue pour offrir un support pour les collections distribuées, en permettant leur création et accès distants. En même temps, des algorithmes nécessaires pour leur utilisation sont proposés, utiles pour le traitement complexe des données. Les collections distribuées proposées gèrent plus exactement la création distante des structures de données multiples, comme les tableaux, les listes ou les maps. Les collections distribuées n'introduisent pas de la fragmentation des données. Leur

[3]Laboratoire d'Informatique Fondamentale de Lille

création distante reste explicite et non transparente pour le développeur, comme le montre l'exemple suivant :

```
// création d'un map distant qui accepte des clés dupliquées
VHashMap map = new VHashMap(true, "localhost:8000");
// création d'une queue distante
VDeque deque = new VDeque("localhost:8000");
```

Les algorithmes attachés à la collection distribuée sont équivalents à ceux offerts pour les collections Java locales, tout en restant séquentiels.

```
// appliquer le même traitement d'affichage sur chaque élément de la queue
VApplying.forEach(deque, new Print(), "locahost:8000");
```

De cette façon, les collections distribuées JGL n'introduisent pas de parallélisme implicite. De plus, le parallélisme n'est exprimé dans aucune de ses formes. L'extension des opérateurs qui peuvent être appliqués est faite grâce au principe de pattern[4] opérateur, en précisant la fonction à invoquer. Le parallélisme peut être ainsi introduit explicitement par le programmeur, par la gestion des threads, comme dans le cas du langage Java.

Le pattern opérateur exige que toute nouvelle méthode, à appliquer sur une donnée, soit définie dans une nouvelle classe, ce qui fait croître le nombre de classes à déclarer, correspondant aux traitements d'un type de données.

## 5.4 DPJ

DPJ [IGD+97], développé à l'Académie Russe des Sciences, est une bibliothèque prévue pour l'utilisation du langage Java dans le développement des programmes parallèles orientés vers les données, sous le modèle d'exécution parallèle SPMD. Les parties séquentielles d'un programme parallèle peuvent être exécutées en parallèle sur des processeurs d'une machine parallèle, les parties étant liées par le standard de communication de messages MPI (Message Passing Interface) [WA99, For03].

Le concept utilisé dans la bibliothèque, qui groupe des ensembles d'objets similaires, est l'*objet conteneur*. Le conteneur distribué représente l'implémentation d'une certaine distribution des données. Celle-ci range ses éléments de telle sorte que chaque nœud du conteneur contienne un élément placé sur le sous-réseau donné à l'instanciation du conteneur distribué.

```
MyDBTree tree = new MyDBTree(Subnet.netWorld);
```

Le placement des conteneurs reste explicite pour le développeur, par la spécification de leur localisation lors de l'instanciation.

Les traitements appliquables sur les éléments des conteneurs ne sont pas encapsulés avec les données, DPJ prenant comme approche le pattern opérateur, similaire à la bibliothèque JGL, mais en introduisant du parallélisme de données. Le traitement invoqué sur les éléments qui implémentent une interface de l'algorithme parallèle, est défini par la méthode *run*, spécifiée par le programmeur et qui représente le corps de l'algorithme exécuté sur chaque nœud.

```
class MyInput extends DUnaryOutputIteratorClass
              implements ParallelAlgorithm {
   ...
   public void run() { /* code de la méthode à appliquer */ }
}
```

[4] modèle

En se basant sur le même pattern opérateur, le programmeur est amené à ne définir qu'une seule opération à appliquer, en parallèle, par classe. DPJ n'exprime aucun parallélisme de tâches.

## 5.5 Ajents

Ajents [ICB99], développé dans le cadre du département d'Informatique aux Universités York, Toronto et Waterloo, Ontario, est une collection de classes de bibliothèque, 100 % en Java, qui implémentent et combinent des caractéristiques importantes, essentielles pour gérer la distribution des programmes parallèles Java. Ces caractéristiques incluent : la possibilité de création à distance des objets, l'interaction avec ces objets de manière synchrone ou asynchrone, et la migration des objets entre des machines hétérogènes.

La création à distance des objets Ajents n'est pas entièrement transparente pour le programmeur. Elle utilise des serveurs dédiés Ajents, qui s'exécutent sur toute machine virtuelle Java appartenant à l'environnement. Le développeur informe tout d'abord le système Ajents de la demande de création d'un objet distant. Suite à cette demande, un ordonnanceur d'objets permet de retrouver un serveur distant de création d'objets, disponible. Ce serveur sera invoqué afin de pouvoir créer à distance, sur la machine du serveur d'objets, un nouvel objet.

```
// enregistrer pour obtenir un ordonnanceur d'objets
// qui connaît des serveurs disponibles
AjentsScheduler sched = Ajents.register();
// créer un objet sur un serveur disponible
AjentsObj obj = Ajents.new("ajents.tests.NewObj", NewObj_1",
                "/home/ajents/tests/myobjs.jar", sched.Availserver());
```

En passant par le serveur de création d'objets, le déploiement d'objets distants est explicite, au niveau de la demande de placement, mais implicite, au niveau de la politique de choix de placement.

Aucune forme de parallélisme de données n'est exprimée. Au contraire, le parallélisme en Ajents est exprimé grâce aux appels asynchrones, sous la forme de parallélisme de tâches. Les méthodes qui peuvent être appliquées sont celles appartenant à la définition de la classe de l'objet. Ces appels sont explicitement synchrones ou asynchrones, exprimés par des invocations statiques de la classe *Ajents* de la bibliothèque :

```
// appels synchrones
try {
  Ajents.rmi(obj, "setName", "NewObj");
} catch(Exception ex) {...}

// appels asynchrones
Future future = Ajents.armi(obj, "getName");
try {
  String name = (String)future.get();
} catch(Exception e) {...}
```

Pour les appels synchrones, les exceptions sont propagées à l'endroit où l'appel RMI a été effectué. Pour les appels asynchrones, les retours sont gérés par un objet futur, aussi bien pour une valeur que pour une exception.

La généricité de l'appel asynchrone, en Ajents, implique la perte de typage fort pour les types des arguments passés en paramètre, situation rencontrée aussi dans le projet Jacob.

## 5.6 Jacob (Java Active Container of Objects)

Jacob [DS , Vig99], développé à LABRI[5], à l'Université de Bordeaux, est un support de développement Java qui cherche à faciliter la distribution d'applications, à travers un outil dynamique, sans introduire un surcoût important. Cet outil de distribution d'applications Java est basé sur le concept de *conteneur actif*. Le conteneur actif est un conteneur d'objets qui permet l'activation des traitements sur les objets qu'il contient. Cette enveloppe offre la possibilité de rendre un objet Java local comme un objet distant, de manière dynamique, grâce à des primitives attachées au conteneur :

- put (Object key, Objet object) - insérer un objet dans le conteneur,
- remove (Object key) - supprimer un objet du conteneur,
- get (Object key) - récupération d'un objet du conteneur.

La distribution des objets se base sur l'insertion des objets dans des conteneurs, action qui reste explicite pour le programmeur. En comparaison avec des outils de distribution existants, statiques et automatiques (où la création distante des objets et la distribution sont réalisées de manière implicite), l'approche Jacob de distribution n'est pas automatique. La distribution est faite explicitement par le programmeur, en conservant la même déclaration de classe de l'objet, mais elle est dynamique.

```
MyObject object = (MyObject)fromActiveContainer.get(key);
fromActiveContainer.remove(key);
toActiveContainer.put(key, object);
```

Une autre primitive est attachée au conteneur, réalisant un appel asynchrone d'une méthode d'un objet du conteneur spécifié par la clé :

```
void call (Object key, String method, Object[] args, MethodResult result);
```

Pour cet appel asynchrone, le résultat (valeur ou exception) peut être récupéré à travers le principe d'objet futur, utilisé aussi dans le projet ProActive.

```
MethodResultImpl res = new MethodResultImpl();
activeContainer.call(key, "compute", args, res);
...
try {
  MyResultObject myRes = (MyResultObject) res.getReturnedResult();
} catch(Throwable t) {...}
```

L'inconvénient d'une telle expression d'appels asynchrones est la perte du typage fort. Cet inconvénient est pallié par une solution d'appel totalement transparent et asynchrone qui utilise des créations dynamiques de proxys. Un appel complètement transparent pose deux problèmes : la gestion des exceptions et l'identification, par l'utilisateur, du type d'appel réalisé, synchrone ou asynchrone. La semi-transparence d'appels asynchrones, offerte par des appels marqués spécifiques, résout les problèmes posés par la transparence totale.

Jacob propose des appels asynchrones, qui expriment un volet du parallélisme, le parallélisme de tâches, sans pour autant proposer l'expression d'un parallélisme de données, ceci n'étant pas prévu dans l'environnement.

[5]Laboratoire Bordelais de Recherche en Informatique

## 5.7 ProActive

ProActive [CKV98, BCHV00], développé à l'INRIA[6] Sophia Antipolis, est une bibliothèque 100 % Java pour la programmation parallèle, distribuée et concurrente. La facilité de programmation ProActive, le déploiement des applications et l'asynchronisme de traitement reposent sur le concept *d'objet actif*. Le pattern particulier d'objet actif proposé, encapsule différentes caractéristiques : l'accessibilité à distance, la création à distance, la transparence de localisation (locale ou distant) et l'asynchronisme d'appel (un propre thread d'exécution est attaché à chaque objet actif), la migration.

Le modèle de programmation en ProActive offre du polymorphisme entre les objets actifs et passifs (locaux) : une application peut contenir à la fois des instances des objets passifs ou actifs d'une classe donnée. Les transformations des objets passifs en objets actifs conservent le polymorphisme, par l'usage d'un même type. Une deuxième solution pour créer des objets actifs en ProActive est l'instanciation, grâce à une méthode statique de la bibliothèque.

```
Object[] params = new Object[]{new Integer(26), "aString"};
A a = (A)ProActive.newActive("example.A", params);
// la création peut prendre comme paramètre auxiliaire un noeud,
//  pour la précision du placement de l'objet
// ou
A a = new A(26,"aString");
a = (A)ProActive.turnActive(a);
```

La sémantique des objets diffère lors de l'accès : un appel sur un objet actif sera asynchrone (sauf pour des cas particuliers mentionnés en [Tea02]), tandis que pour un objet passif, l'appel est synchrone. L'appel asynchrone sur un objet passif s'exécute dans son propre thread d'exécution, lancé à la création, de manière transparente pour l'utilisateur. Les différents appels asynchrones arrivant sur l'objet sont sauvegardés et desservis en fonction de la politique FIFO. Ce comportement, qui peut être changé par la spécification des activités à réaliser avant et après l'exécution de l'appel, définit une synchronisation intra-objet. L'existence d'un thread par objet actif exprime un parallélisme de tâches, mais son unicité limite le degré d'asynchronisme sur un objet actif.

Un deuxième type de synchronisation, l'attente par nécessité, concerne l'interaction entre des objets : la synchronisation inter-objets. Chaque appel asynchrone renvoie un objet futur, à condition que l'objet soit réifiable [BCHV00]. Un objet futur est un objet non-actif, qui accueille les résultats des méthodes invoquées et pas encore terminées.

La communication point-à-point offerte en ProActive, entre les objets actifs, est enrichie avec un mécanisme de communication de groupe [BcBC02], qui utilise la même technique de réification et de protocole meta-objet, comme pour les communications point-à-point. Les groupes sont constitués des objets actifs ou passifs, d'un type minimal donné. Des objets déjà créés peuvent joindre un groupe. L'appel vers un groupe diffuse la méthode correspondante vers tous les éléments du groupe, conservant la sémantique imposée par les types d'éléments : si l'élément est un objet actif, l'appel est asynchrone, sinon l'appel est un appel Java local. Ce type de communication permet d'exprimer un parallélisme de données sur les éléments d'un groupe. Le résultat d'une communication de groupe est aussi un groupe, sur lequel différentes méthodes facilitent la synchronisation (attente d'un premier élément du groupe résultat, ou de tous les éléments, etc.).

Les communications de groupes sont typées : uniquement les méthodes appartenant aux classes ou aux interfaces implémentées par les éléments du groupe peuvent être appelées, la vérification

[6]Institut National de Recherche en Informatique et en Automatique

étant réalisée à la compilation.

Le placement des objets actifs en ProActive est explicite : un paramètre supplémentaire donné lors de la création indique le nœud accueillant l'objet. Un nœud permet de contrôler le mapping des objets, par défaut, les créations étant locales à la machine d'appel. Afin de se dispenser de la référence des nœuds (associée à un nom symbolique donné par un URL), le conception de *nœud virtuel* est utilisé [cBCH+02]. Un nœud virtuel est une entité logique sur laquelle les objets actifs sont déployés.

Le mapping des nœuds virtuels sur les nœuds et les JVM est réalisé grâce à un fichier de description. Un contrôle de placement des objets est également acquis par un monitoring graphique de l'application : des créations et des migrations d'objets actifs peuvent être exécutées. Un objet actif peut migrer si une méthode d'activation de la migration est définie par l'utilisateur, qui appelle une des primitives de migration ProActive, et retourne toute de suite. Cette sémantique conserve l'objet actif dans un état indépendant de la pile d'exécution (il n'existe pas de méthode en cours d'exécution sur l'objet, autre que la demande de migration).

De plus, l'environnement graphique fournit des informations liées à l'exécution de l'application (les liens de communication entre les objets, l'état des objets : en attente ou en exécution).

## 5.8 Do!

Do! [LP00], développé dans le projet PARIS, à l'IRISA[7] de Rennes, est un environnement composé d'un framework parallèle, d'un framework distribué et d'un pré-processeur, qui coopèrent afin de faciliter la maîtrise de la conception d'applications distribuées et leur déploiement.

La distinction entre le code parallèle et le code réparti, suggérée dans la méthodologie de développement, est réalisée de la manière suivante : le programmeur écrit d'abord l'application sous forme parallèle, puis il y introduit des spécifications de distribution.

La déclaration de code parallèle se réalise dans le framework parallèle en définissant certains composants (en héritant des classes). Le programmeur peut décrire, au moyen de ce framework, le schéma d'exécution des tâches, d'accès aux données ou de coopération entre les composants d'un programme. Le framework parallèle Do! suppose l'implication du développeur dans la définition des composants manquants, pour spécifier les calculs de ses applications.

Une autre approche d'aide à la conception, proposée en ADAJ, est la bibliothèque comme ensemble de composants réutilisables, qui aide le programmeur à maîtriser des fonctionnalités d'ordre général.

Le parallélisme exprimé en Do! est réalisé au travers du concept d'*objet actif*, auquel est associé un flot de contrôle. La définition d'un flot de contrôle est similaire à la notion de pattern opérateur, permettant la déclaration d'une seule tâche par classe.

```
public class TASK {
  // définition du comportement de la tâche
  protected void run (OBJECT param) {
    while (true) action (param);
  }

  protected void action(OBJECT param) {return ;}

  // activation synchrone
```

[7] Institut de Recherche en Informatique et Systèmes Aléatoires

```
  public final void call (OBJECT param) {...}
  // activation asynchrone
  public final void start (OBJECT param) {...}
  ...
}
```

Plusieurs flots de contrôle peuvent s'exécuter simultanément dans des objets distincts, modèle exprimant un parallélisme *inter-objets*. Dans un même objet, plusieurs flots de contrôle peuvent aussi s'exécuter simultanément, ce qui est réalisé en groupant des objets actifs dans des collections. Ce type de parallélisme est nommé *intra-objets*.

Do ! est moins orienté vers le déploiement des applications, lequel, ici, est automatique et transparent. La distribution des objets est réalisée par des instanciations explicites qui précisent la localisation des objets.

```
OBJECT local, distant;
// création locale
local = new OBJECT();
// création distante d'objets
distant = DoRuntime.remoteNew(nodeId, "OBJECT", argArray);
```

Les éléments d'une collection sont créés avant l'introduction dans la collection, donc leur placement est décidé à l'instanciation sans possibilité de migration par la suite.

## 5.9 Bilan

Les différents travaux portant sur la mise en œuvre des environnements de développement des applications parallèles, des bibliothèques ou frameworks parallèles sont réalisés avec l'objectif de permettre, au développeur, d'écrire des programmes parallèles de manière simple et intuitive. Le compromis entre l'expressivité des langages et l'utilisation efficace des systèmes d'exécution impose la mise en œuvre des mécanismes de déploiement des applications parallèles.

Les politiques de placement sont intégrées lors de la conception des applications orientées objets et distribuées, à l'aide de la création des objets et de leur accès à distance. Plusieurs projets (Ajents, DPJ, ProActive, JGL et Jacob) proposent un placement explicite des objets, soit par la spécification de la localisation, soit par des requêtes auprès des services de création d'objets, capables de produire la meilleure solution de placement. D'autres projets proposent un placement implicite, par défaut, en utilisant une certaine politique, comme dans le projet Do !. L'environnement proposé par JavaParty donne le choix entre un placement implicite (politique par défaut) et un placement explicite.

L'accès transparent aux objets distants est acquis à travers des transformations de code (JavaParty, Do !) ou par génération des talons particuliers (ProActive). D'autres projets conservent la différence entre un appel distant et un appel local, en utilisant le pattern opérateur (JGL, Jacob) ou en récupérant des exceptions supplémentaires (Java RMI).

Le parallélisme est souvent exprimé dans les projets cités, sous la forme de parallélisme de tâches, par le biais d'invocations asynchrones (Ajents, DPJ, Jacob) ou de concept d'objet actif (Do !, ProActive).

L'expression d'un parallélisme de données est obtenue, et c'est ce qui lui donne l'intérêt, par la bibliothèque DPJ, ou par le projet Do !, par le groupe d'objets actifs. Le degré et la granularité du parallélisme sont gérés de manière explicite dans tous les projets le traitant.

Ces outils sont intégrés dans les environnements soit en utilisant, à 100 %, le langage Java (Ajents, ProActive, Jacob), soit à l'aide de pré-processeurs (JavaParty, Do !) (voir tableau 5.1).

| | Dome | ACADA | JGL | DPJ |
|---|---|---|---|---|
| *distribution objets* | explicite (directives) | explicite | explicite (param) | explicite (param) |
| *transparence accès* | oui | - | non | non |
| *expr // tâches* | non | oui | non | non |
| *expr // données* | oui | oui | non | oui |
| *degré et granularité* | - | dynamique (refragmentation) | - | - |
| *Java %* | 0 % | 0 % | 100 % | 100 % sans MPI |

| | Ajents | ProActive | Jacob | Do ! | JavaParty |
|---|---|---|---|---|---|
| *distribution objets* | explicite (sched) | explicite (monitoring/ f. descript) | explicite | implicite (modèle de distrib) | implicite aléatoire (par défaut) |
| *transparence accès* | non | oui | non | oui | oui |
| *expr // tâches* | oui | oui | oui | oui | non |
| *expr // données* | non | oui | non | oui | non |
| *degré et granularité* | - | degré limité d'asynch | - | explicite | explicite |
| *Java %* | 100 % | 100 % | 100 % | 100 % sans précomp | 100 % sans précomp |

Tableau 5.1 – Bilan des projets

Une autre possibilité est d'utiliser un autre mécanisme de communication distante, que celui de Java, celui de passage de messages (MPI dans la bibliothèque DPJ).

# Chapitre 6

# L'expression du parallélisme en ADAJ

De manière générale, le parallélisme peut être exprimé au travers de l'activation de calculs parallèles (parallélisme de tâches) ou au travers de la distribution de données (approche SPMD). L'environnement ADAJ permet d'exploiter et de combiner ces deux approches. Les deux techniques pour obtenir le parallélisme en ADAJ sont présentées dans les sections 6.1 et 6.2. Leur présentation impose le détail sur la structure générale et les fonctionnalités attachées (les traitements, la gestion de la synchronisation, ou des erreurs).

## 6.1 Le parallélisme de données au travers du concept de collection distribuée

Les objectifs du projet ADAJ ne peuvent être atteints ni par l'utilisation des bibliothèques JGL du projet Voyager de ObjectSpace [Obj96], qui ne profitent pas de l'aspect multithreadé de Java, ni par le regroupement sous forme de vecteur de JavaParty (dans sa première version 0.98), qui n'offre aucun outil pour les traitements sur les données, tels que ceux proposés sur les collections Java ([SUNc]). Les vecteurs en JavaParty sont similaires à ceux de Java, mais offrent une implémentation distribuée et permettent ainsi de représenter des ensembles de données distribuées. Toutefois, ils conduisent à un éparpillement des données, source d'inefficacité lors de l'exécution des traitements.

La notion de groupe de ProActive (voir la section 5.7), réunissant des objets actifs (et passifs), permet à la fois une expression d'un parallélisme de données, par l'exécution d'un même traitement sur les éléments d'un groupe (grâce à la communication de groupe), et une expression d'un parallélisme de tâches, par l'activation asynchrone des objets actifs. La sémantique particulière des objets actifs impose certaines contraintes dans l'utilisation des groupes (les éléments d'un groupe sont monothreadés, leur types ne peuvent pas être finaux). Le typage des groupes rend la syntaxe des communications de groupe transparente : l'utilisateur n'est plus conscient du type d'appel réalisé (asynchrone ou de groupe) à cause de l'identification du type du groupe avec le type de ses éléments. ADAJ vise à fournir une transparence syntaxique, mais partielle (voir la section 4.2.3), afin que la volonté du développeur de l'écriture parallèle soit exprimée explicitement.

Le parallélisme de données, inspiré d'un parallélisme de tâches, se retrouve également dans le projet Do! (voir la section 5.8), sous forme de pattern opérateur : les données et les tâches sont des entités réunies séparément dans des collections, fournies à des constructeurs parallèles.

Dans une première tentative en ADAJ [FTD01], le concept d'opérateur a été repris, mais des contraintes sémantiques (objets remote et locaux, ou éloignement d'un modèle séquentiel) nous ont

déterminé à considérer une autre approche, présentée par la suite.

ADAJ introduit une structure hiérarchisée, appelée *collection distribuée*, regroupant des objets dans des *fragments* ; les fragments sont distribués et des traitements parallèles activés sur des fragments réalisent le parallélisme de données selon une approche SPMD. Les fragments constituent l'expression de la granularité du parallélisme, aspect détaillé dans la section 6.1.1. L'idée d'exploitation du parallélisme de données à travers les données fragmentées a initialement été introduite dans le projet ACADA [VDL98], implémenté en langage C++.

ADAJ associe à ces collections distribuées des opérateurs permettant l'activation parallèle et asynchrone de méthodes sur les fragments (voir la section 6.1.2) et assure, au travers d'*objets futurs* (voir la section 6.1.3), une récupération différée et asynchrone des résultats. L'utilisation des collections distribuées s'inscrit ainsi dans une méthodologie de programmation pour l'expression des traitements parallèles.

### 6.1.1 La hiérarchie de la collection distribuée

Les collections existant en Java permettent de rassembler un nombre quelconque de données dans des groupes, sur lesquels peuvent être lancés des traitements séquentiels [SUNc]. En intégrant ce concept dans un environnement distribué, ADAJ propose une structure hiérarchique, *la collection distribuée*, dont la racine donne accès aux *fragments* situés sur le deuxième niveau, les fragments contenant des objets. Le terme de collection fragmentée serait plus approprié mais nous conservons l'appellation plus classique de collection distribuée.

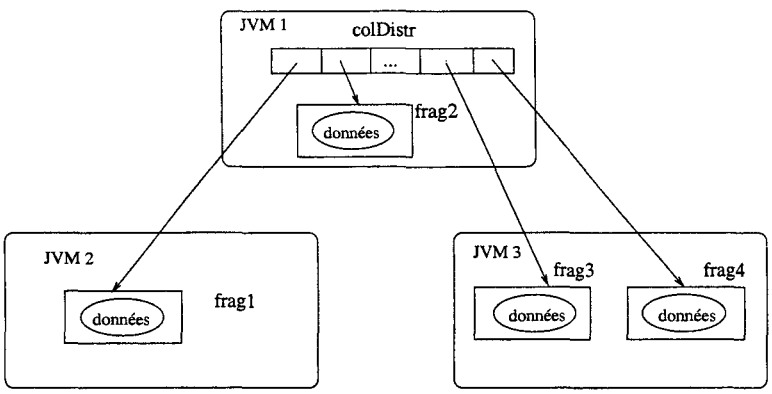



Figure 6.1 – Organisation de la collection distribuée

La figure 6.1 illustre la structure d'une collection distribuée, *colDistr*, qui contient les fragments (*frag1*, *frag2*, *frag3* et *frag4*), instanciés respectivement sur les machines virtuelles Java (JVM) 2, 1, 3 et 3, les fragments étant des objets qui contiennent des données (i.e. des objets). Nous remarquons qu'il peut y avoir plusieurs fragments sur la même machine virtuelle et la collection distribuée et les fragments peuvent coexister sur une même JVM.

#### La collection distribuée

La collection distribuée est un objet remote, au sens JavaParty, tenant compte de son accessibilité à distance. Sa gestion coïncide, de manière fonctionnelle, avec celle d'un objet de type collection Java classique, en permettant :

- d'accéder aux informations sur sa taille (le nombre de fragments),

- de retrouver un fragment ou l'indice d'un fragment donné (sa position dans la collection distribuée),
- de manipuler les fragments à partir de leur indice,
- d'itérer sur les fragments.

Lors d'un traitement parallèle sur une collection distribuée, la structure de ses fragments peut varier. Si deux collections distribuées partagent un même fragment, de coûteux mécanismes de synchronisation devraient être mis en œuvre pour assurer un bon fonctionnement. Pour résoudre ce problème de partage et pour respecter en même temps la sémantique de la structure hiérarchique de la collection distribuée, ADAJ exige qu'un fragment ne soit rattaché qu'à une seule collection distribuée.

Un état cohérent d'une collection distribuée est aussi acquis par une gestion particulière des fragments. La création d'un fragment, à travers la méthode *addFragment* de la classe *DistributedCollection* (la classe des collections distribuées), garantit le non-partage d'un même fragment par plusieurs collections distribuées ; pour respecter cette contrainte, il n'y a pas de méthode permettant à l'utilisateur d'ajouter, à une collection distribuée, des fragments qui ont été créés à l'extérieur de cette collection distribuée. Ainsi, la création d'un fragment peut uniquement être réalisée, soit à travers une méthode appartenant à la collection distribuée, qui réalise en même temps l'ajout de ce nouveau fragment dans la collection, soit à travers le constructeur de la classe *DistributedCollection*.

### Le fragment

Le fragment est également un objet remote qui encapsule d'autres objets. Pour l'efficacité des traitements parallèles, il est préférable que les objets des fragments soient sur la même machine virtuelle que le fragment, pour donner le maximum de localité aux traitements. Les objets d'un fragment seront donc des objets locaux (au sens JavaParty). Lors de la migration d'un fragment (qui est un objet remote), les objets locaux de ce fragment seront donc des objets "nouveaux", copiés dans la machine destinataire, et ils n'engendront pas de communications à distance.

La couche supérieure, la collection distribuée, qui réunit les fragments, permet une communication entre les fragments, vue la notion d'ordre qui existe entre eux. Ainsi, un fragment a connaissance des fragments voisins et de tout fragment d'un indice donné.

Le nombre de fragments n'est pas nécessairement fixé lors de la conception du programme. Il peut être déterminé lors de l'exécution, de manière dynamique, par ajouts successifs de fragments. Le programmeur peut aussi fixer au départ le nombre de fragments par une création implicite lors de la construction de la collection distribuée (les fragments étant vides au départ). L'utilisateur maîtrise de cette façon le degré du parallélisme, en fonction des paramètres de l'exécution et des caractéristiques de la plate-forme d'exécution disponible. La granularité du parallélisme est également gérée par l'utilisateur, qui est seul à connaître la nature de l'application. Le fragment lui offre la possibilité de gérer le parallélisme, et éventuellement de l'adapter (par des ajouts ou suppressions des fragments de la collection distribuée).

Le type de distribution des fragments peut etre soit aléatoire, suivant le type de placement pour tout autre objet remote, soit spécifique (par bloc ou cyclique). Lors de la création de la collection distribuée, ce type de distribution peut être spécifié, grâce à un paramètre auxiliaire. La flexibilité de placement des entités distribuées se retrouve aussi dans le projet Do!, d'une manière différente : un modèle de distribution est attaché à chaque collection, qui met en œuvre la distribution par défaut de la collection. Cette spécification pourrait être changée dynamiquement, avec le risque de perdre la cohérence de la distribution. L'association de cette spécification au constructeur de la collection distribuée en ADAJ élimine ce risque.

Le placement des fragments sur les machines de la grappe et le mécanisme explicite de migration de JavaParty ne sont que des outils de base pour la distribution des objets. Une telle distribution ne peut pas s'adapter aux modifications des charges des machines (ou, généralement, du support d'exécution), ou aux évolutions des calculs. Pour pallier cet inconvénient, en ADAJ, les fragments se soumettent à la politique implicite de distribution qui se base sur des informations dynamiques (les relations entre les objets) lors de la création ou d'une redistribution déclenchée par le mécanisme d'équilibrage de charge. Le regroupement des objets dans les fragments reste explicite et la structure de la collection distribuée permet de contrôler la granularité du parallélisme, au travers de la taille des fragments, en remplaçant l'activation d'un traitement parallèle sur tous les objets par l'activation de traitements parallèles sur les fragments d'objets (voir des exemples dans le chapitre 8).

Dans le cadre d'ADAJ, en se basant sur le mécanisme de migration de JavaParty, nous nous intéressons à intégrer un outil différent de migration, spécifique aux fragments. Des fragments actifs (pour lesquels une méthode est en cours d'exécution) pourront ainsi être migrés, ce qui n'est pas possible dans la version courante de JavaParty. Ce mécanisme de migration est détaillé dans la section 14.3.4.

Le fragment ajoute une fonctionnalité supplémentaire à l'expression du parallélisme de données (existant dans d'autres projets, mais plus aisément exprimée en ADAJ), qui concerne sa participation au mécanisme d'équilibrage de charge en tant qu'objet migrable, par un placement adéquat ou une migration.

### 6.1.2 Distribution du traitement

#### Activation des traitements parallèles

Dans l'environnement ADAJ, l'utilisateur dispose d'une primitive *distribute* comme outil d'expression du parallélisme de données qui permet de diffuser un traitement à chacun des fragments et d'assurer la récupération des résultats. La primitive distribute active de manière asynchrone une même méthode spécifiée sur les fragments.

Les primitives distribute doivent désigner la méthode à appliquer et permettent la récupération des résultats, ou l'attente synchrone dans le cas de méthodes qui ne renvoient pas de résultat (la syntaxe exacte est décrite dans la section 7.3).

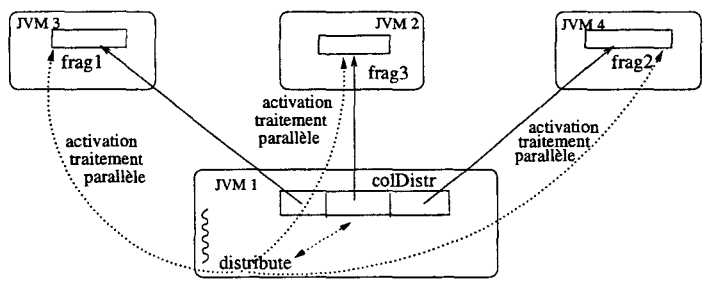



Figure 6.2 – Activation d'un distribute

Une primitive du type distribute considère :
- la collection distribuée sur laquelle elle s'applique,
- le traitement parallèle à exécuter sur chacun des fragments (le même pour tous les fragments),
- les paramètres nécessaires pour le traitement, qui peuvent être les mêmes pour chaque fragment (primitives du type *distribute*) ou différents (primitives du type *distributeD*),

- une structure particulière, appelée *collecteur*, qui permet la récupération des résultats (structure précisée dans la section 6.1.3).

La figure 6.2 présente un exemple de distribution d'un traitement parallèle ; une méthode distribute est activée sur la collection distribuée *colDistr* située sur la JVM1, ce qui a pour effet d'activer de manière asynchrone un même traitement sur chaque fragment de la collection distribuée.

Plus précisément, l'exécution de la primitive distribute sur une collection distribuée provoque l'activation de threads locaux $T$ (un par fragment) associés aux différents fragments (figure 6.3). Chaque thread $T$, appelé *thread de transition*, permet l'activation de threads RMI distants, appelés *threads de traitement*, chargés d'exécuter le traitement sur chacun des fragments, et attend le retour du résultat correspondant.

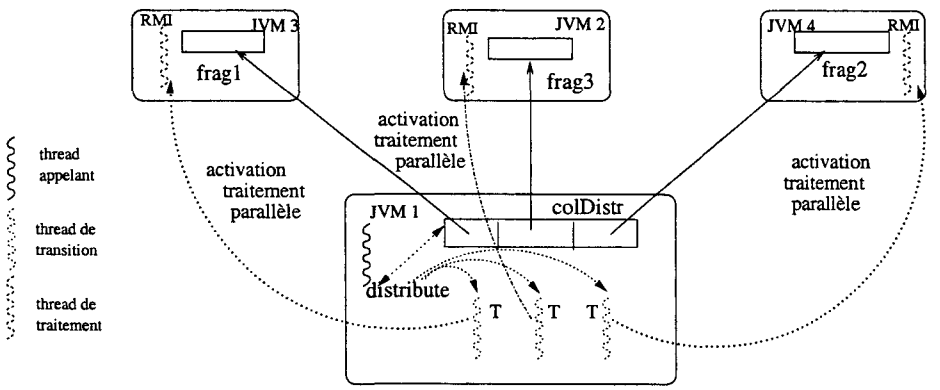



Figure 6.3 – Activation d'un distribute (2)

L'inconvénient des appels de type distribute réside dans le non-typage de la méthode exécutée de manière parallèle, par rapport à sa déclaration. La raison est l'utilisation de l'introspection (voir la section 7.2) pour la recherche de la méthode à appliquer. Ainsi, lors d'une construction inadéquate d'un appel de distribute, l'utilisateur n'en sera averti qu'à l'exécution, pas à la compilation. Cet aspect sera écarté, par une vérification de type lors d'un appel de distribute, grâce au mécanisme de transparence d'appels (similaire à [Vig99]), dans une deuxième solution d'implémentation de la bibliothèque (voir la section 7.5).

### Synchronisation

Des problèmes de synchronisation peuvent aussi apparaître si, lors d'un distribute, les traitements parallèles consistent à changer l'organisation de la collection (des ajouts ou des suppressions de fragments). La collection distribuée ne conserve plus le même état, ce qui peut dérouter l'utilisateur. La solution proposée est de réaliser une copie des fragments (des références) juste avant le lancement de distribute. Les traitements parallèles activés correspondent, ainsi, à un état cohérent de la collection distribuée, comme une image figée.

La copie de la table de fragments (voir la figure 6.4) a également l'avantage d'optimiser le nombre d'appels distants pour la gestion des traitements parallèles. Les références des fragments (indication 1 de la figure 6.4) sont récupérées, une fois pour tout appel distribute (opération 0), par le biais d'un seul appel distant vers la collection distribuée. Dans la même figure, l'opération 2 réalise l'activation des traitements sur chacun des fragments, qui retournent leur résultat (opération 3), mis dans le collecteur à travers l'opération 4.

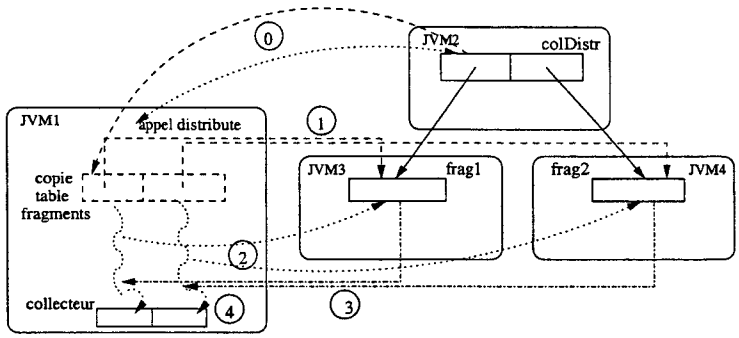



Figure 6.4 – La synchronisation des appels distribute

### 6.1.3 Récupération du résultat

#### Principe

L'exécution d'une primitive distribute sur une collection distribuée provoque l'activation de méthodes en parallèle qui entraînent le retour de plusieurs résultats, souvent dans un ordre quelconque, différent de l'ordre des activations des traitements sur les fragments. Il impose donc l'introduction d'une structure d'accueil pour la collecte et la gestion des résultats : *le collecteur*.

Un collecteur est créé à chaque appel d'un traitement parallèle, donc il est propre à l'exécution d'une primitive distribute. Cet objet agit comme un objet temporaire d'une invocation de méthode non encore terminée. Le collecteur sera mis à jour au fur et à mesure des retours des résultats : les traitements lancés pour chaque fragment, une fois terminés, mettent leur résultat dans le collecteur. Le même concept, sous forme d'objet futur, est utilisé dans le projet ACADA [VDL98], et repris, plus tard, dans les projets Ajents [ICB99] et ProActive [BCHV00].

Le collecteur a le rôle de gestionnaire des résultats (pour les traitements qui retournent un résultat) ou permet l'attente synchrone de la fin des traitements (pour les traitements qui ne renvoient pas de résultats).

Pour une méthode sans retour de résultat, le lancement se fait de manière asynchrone et, à travers la primitive *waitEnd*, l'utilisateur peut s'assurer de la terminaison des exécutions sur chaque fragment. Cette fonctionnalité attachée au collecteur a un rôle équivalent à une barrière de synchronisation.

L'utilisateur qui a lancé une exécution parallèle par un distribute dispose d'un ensemble de primitives permettant de gérer les retours asynchrones : l'attente de tous les résultats (*getAll*), l'obtention d'un résultat disponible, rangé dans le collecteur et pas encore traité (*getOne*), ou la récupération d'un résultat issu d'un fragment particulier (*getI*).

Les primitives d'accès aux résultats renvoient un seul objet, pour *getOne* et *getI* et un tableau d'objets pour *getAll*. Le résultat contient des informations supplémentaires concernant la référence du fragment sur lequel le traitement a été lancé et son indice dans la collection distribuée. Ces informations sont ajoutées pour informer l'utilisateur de l'origine des résultats et pour éviter de nouvelles communications éventuelles relatives à la recherche du fragment pour lequel le résultat a été obtenu.

### Organisation

Le fonctionnement général d'un collecteur est le suivant : les méthodes invoquées retournent leurs résultats aux threads de transition, qui les avaient activées et qui sont bloqués en attente du retour de résultat. Chaque thread de transition range le résultat reçu dans le collecteur (figure 6.5) dès qu'il le reçoit.

Les résultats issus d'un appel distribute sont à utiliser localement, dans la même machine que l'appel. En conséquence, pour éviter des appels distants lors de l'exploitation des résultats, le collecteur est un objet local. La création locale des threads de transition, intervenant dans un distribute, permet de mettre à jour le collecteur, avec des résultats partiels.

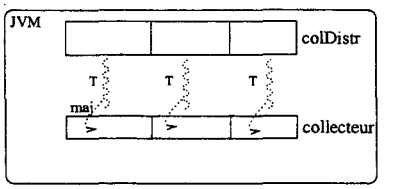



Figure 6.5 – Récupération des résultats

Le choix d'un collecteur local (et en conséquence sa localisation sur la machine appelante) minimise les communications (au sens communications distantes).

Pour gérer les résultats issus des traitements lancés par un distribute, le collecteur est implémenté comme un tableau. Cette structure est enrichie avec un nombre d'informations utiles, pour offrir les fonctionnalités de base attachées au collecteur.

Une première fonctionnalité offerte par le collecteur est la récupération de tous les résultats de distribute. Ainsi, un compteur de résultats disponibles est incorporé au collecteur. Sa valeur est comparée avec le nombre total de résultats attendus. En cas d'infériorité, la méthode est bloquée. Le déblocage est effectué lors de l'arrivée d'un nouveau résultat, et le test est répété.

Dans le cas des méthodes qui ne renvoient pas de résultats, l'attente de la terminaison de tous les traitements est réalisée de manière semblable, sauf qu'il n'y a pas de retour de résultat.

Pour la récupération des résultats spécifiques (le premier ou celui issu du fragment d'un indice donné), une nouvelle structure de données est nécessaire, à l'intérieur du collecteur, pour la raison suivante : la disponibilité et récupération d'un résultat se modélisent comme un problème producteur-consommateur [SGG01]. Le thread de transition récupère le résultat, donc il fournit une nouvelle valeur, et l'appel au collecteur, pour la récupération d'un premier résultat disponible et non traité, réalise la consommation de la valeur. Cette valeur ne sera plus disponible pour un deuxième appel du même type. A l'inverse, elle pourra être retournée si elle correspond au résultat issu d'un fragment particulier, résultat qui a été demandé explicitement par la primitive *getI*. Etant donné que la récupération d'un résultat ne devrait pas bloquer l'arrivée d'un autre, le mécanisme correspond au problème producteur-consommateur, avec un buffer de taille limitée, égale à la taille de la collection distribuée. La nouvelle structure, le buffer, est un tableau, qui dispose de deux méthodes synchronisées *read* et *write*.

L'organisation du collecteur, imposée par la gestion des résultats et les primitives offertes, est la suivante (voir aussi la figure 6.6) :

- un tableau de résultats, *elementData*,
- un compteur, dont la valeur définit le nombre de résultats disponibles, *nb*,
- un buffer de résultats déjà traités, *result*, sous forme d'indices des fragments.

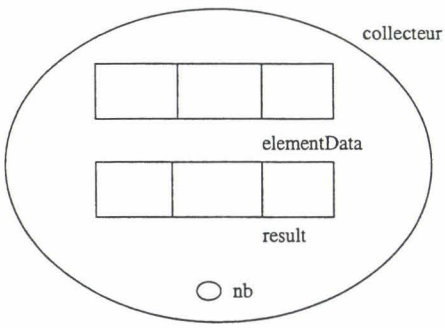



Figure 6.6 – L'organisation du collecteur

La figure 6.7 montre un exemple d'interaction entre les primitives *getI* et *getOne*. Un premier appel, *getOne*, consomme un résultat issu du fragment d'indice 1, indice donné par le premier élément du buffer *result*. L'appel suivant, *getI(1)*, tente à récupérer le résultat issu du même fragment que précédemment, d'indice 1. Etant disponible, il sera retourné directement à l'appelant, par l'accès direct à la structure *elementData*. Un troisième appel, *getOne*, tente à obtenir un résultat disponible, mais pas encore traité. Ce n'est pas le cas pour le résultat du fragment d'indice 1, car il a été déjà utilisé, donc, la structure *result* sera de nouveau accédée, et dès qu'une valeur sera disponible, dans l'exemple, l'indice 2, la valeur correspondante dans la structure *elementData* sera retournée.

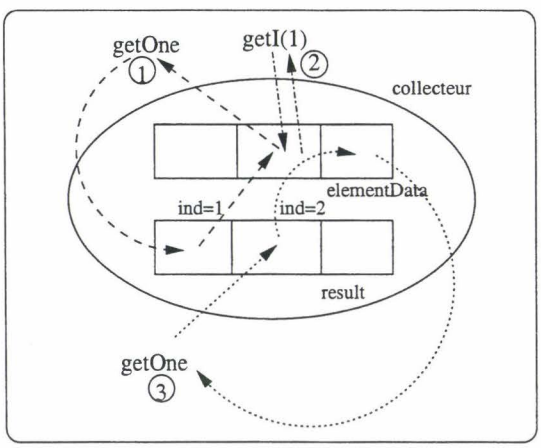



Figure 6.7 – L'interaction des primitives *getI* et *getOne*

Les synchronisations imposées dans le processus de mise à jour des données sont réalisées grâce au mécanisme de synchronisation Java. L'attente d'un résultat, et l'information sur sa disponibilité, sont implémentées à l'aide des outils d'attente (*wait*) et de notification (*notify, notifyAll*) des threads Java ([Sunf]), pour éviter une attente active.

### 6.1.4 Exceptions lors des appels du type distribute

Le traitement des erreurs générées à l'exécution est réalisé grâce au mécanisme d'exceptions. Une exception est un événement, qui se produit pendant l'exécution d'un programme, et qui interrompt le flot normal d'instructions. Les exceptions peuvent être interceptées et traitées.

Pour un appel asynchrone, l'exception ne peut pas être captée, parce que le programme appelant

ne détient plus le contrôle sur la méthode qui s'exécute. Ainsi, l'exception risque d'être perdue pour le flot de contrôle principal. Le problème apparaît donc pour des appels du type distribute, qui se basent sur le principe d'appel non-bloquant (asynchrone).

Le seul contrôle existant lorsque des appels de type distribute ont été lancés, est celui réalisé à travers l'objet futur retourné par ces appels. Ainsi, l'objet futur collecte aussi les résultats et les exceptions possibles.

La récupération des exceptions dans le cas des appels du type distribute considère des exceptions multiples possibles. Etant donné que plusieurs traitements sont exécutés simultanément, et que certains traitements pourraient échouer, plusieurs exceptions peuvent être retournées dans le collecteur.

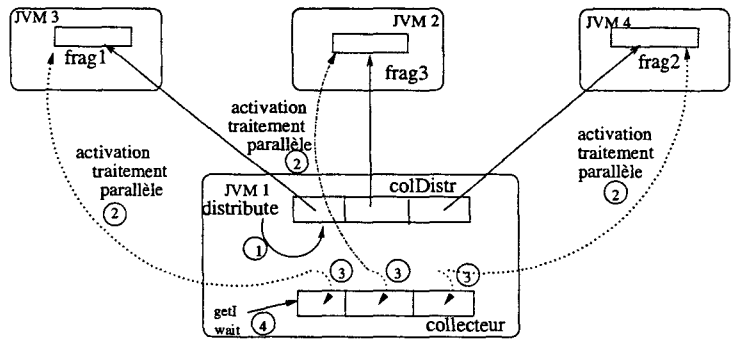



Figure 6.8 – Traitement des exceptions pour un appel distribute

Deux solutions existent pour traiter les exceptions qui apparaissent pendant un appel distribute :
- les exceptions sont accumulées au fur et à mesure,
- une des exceptions est remontée et sauvegardée.

La première solution permet d'offrir à l'utilisateur un historique des exceptions survenues pendant l'exécution du distribute, tandis que la deuxième informe seulement sur l'occurrence d'une exception lors d'un traitement parallèle. Dans tous les cas, il n'y a pas de reprise du traitement ; c'est à l'utilisateur de décider.

En ADAJ, le distribute qui est lancé (opération 1 dans la figure 6.8) invoque le traitement sur chacun des fragments (opération 2). Chaque traitement retourne un résultat, ou rien, ou une exception est levée (le mécanisme de base est présenté dans la gestion des exceptions pour un appel asynchrone dans la sous-section 6.2.3). Ces informations sont mises à jour dans le collecteur (opération 3) et un accès au collecteur (opération 4) récupère l'information.

## 6.2 Le parallélisme de tâches au travers d'appels asynchrones

En Java RMI, les appels de méthodes d'objets distants sont bloquants. Il est nécessaire de recourir au concept de multithreading pour assurer l'activation de calculs parallèles.

Le concept de multithreading est une notion de base pour le modèle de programmation à parallélisme de contrôle. Il permet la création de plusieurs flots de contrôle séparés qui pourront être exécutés sur des processeurs différents. En même temps, ce mode de programmation soulève des problèmes en termes de synchronisation des tâches concurrentes et d'échange d'informations entre ces tâches.

Les techniques de multithreading et d'activation à distance sont exploitées dans le traitement des collections distribuées et permettent un parallélisme d'objets similaire au parallélisme de données.

Mais ADAJ autorise également l'expression d'un parallélisme de tâches classique, comme dans les projets Java Jacob [DS , Vig99] (conteneur actif) et ProActive PDC [CKV98] (objet actif), qui provient des appels asynchrones sur des objets remote, qu'ils soient des fragments ou non.

### 6.2.1 Asynchronisme du traitement

L'asynchronisme du traitement en ADAJ provient des appels asynchrones sur des objets remote. L'asynchronisme d'un appel permet à l'appelant d'exécuter l'invocation suivante sans attendre la terminaison de l'appel en cours.

Pour obtenir l'asynchronisme d'un appel, deux voies sont possibles, basées sur le principe de multithreading : rendre l'appelant multithreadé ou rendre l'appelé multithreadé. Dans le premier cas, l'appelant gère un flot d'exécution supplémentaire pour l'appel voulu asynchrone. Ainsi, l'appelant peut continuer l'exécution sans être bloqué sur l'attente de l'exécution de l'appel. Dans le deuxième cas, l'appelé gère le flot supplémentaire, en simulant un serveur exécutant des services simultanément. Ainsi, l'appelant est uniquement bloqué pendant la transmission de l'appel et peut poursuivre l'exécution sans attendre la fin du traitement invoqué.

De manière similaire au cas de l'appel distribute, le problème du retour du résultat se pose. Le concept d'objet futur est aussi utilisé, accueillant un résultat possible ou permettant de s'assurer de la fin du traitement.

Un cas particulier d'appel asynchrone peut être réalisé sur les fragments, qui sont aussi des objets remote. Des traitements asynchrones peuvent être invoqués, sur un fragment particulier, de l'extérieur de la collection distribuée ou à partir d'un autre fragment (voisin ou d'indice donné).

### 6.2.2 Organisation : asynchronisme client versus asynchronisme serveur

Les deux types de parallélisme exprimés en ADAJ (d'objets et de méthodes) se basent sur le même mécanisme de traitement asynchrone. L'activation des traitements parallèles est réalisée à l'aide des threads. Pour obtenir l'asynchronisme d'un appel distant, les voies possibles citées précédemment sont : activer un thread qui traite l'appel distant dans le site de l'appelant (appelé aussi *asynchronisme du côté client*), ou faire un appel à distance qui consiste à activer un thread qui réalise le traitement (appelé *asynchronisme du côté serveur*).

Le premier mécanisme est montré dans la partie gauche de la figure 6.9. L'appel bloquant de la méthode $m$ est emballé dans un thread (l'équivalent d'un thread de transition), l'initiateur de l'appel étant maintenant le thread de transition. Le résultat de l'appel de la méthode sera récupéré, à l'aide d'un objet futur, mis à jour par le thread. Le deuxième mécanisme, d'asynchronisme de côté serveur, est illustré dans la partie droite de la figure 6.9. L'appelant est conservé, tandis que l'appelé est remplacé avec un nouvel appelé, qui est un thread exécutant la méthode invoquée. Essentiellement, la différence entre les deux méthodes consiste dans la fonctionnalité de ce nouveau thread de transition créé : l'activation de traitement, dans une approche asynchronisme client, et l'exécution du traitement, dans une approche asynchronisme serveur.

En ADAJ, entre les deux types d'asynchronisme d'appels, l'asynchronisme du côté client a été choisi, pour faciliter la récupération des résultats.

La syntaxe d'un appel asynchrone est présentée à l'aide d'un exemple dans la section 7.3.

### 6.2.3 Exceptions lors des appels asynchrones

L'appel asynchrone en ADAJ est un appel semi-transparent du côté client.

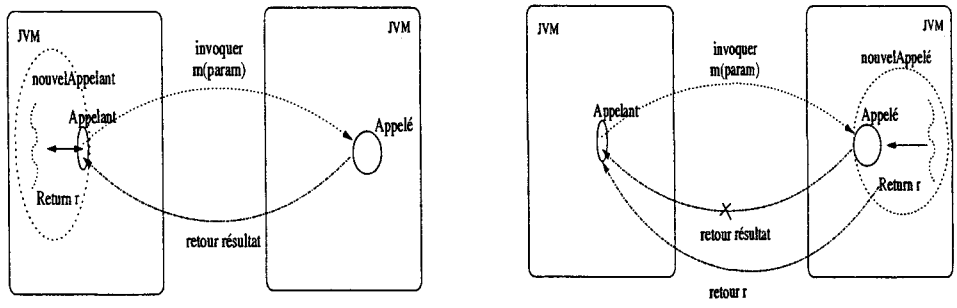



Figure 6.9 – Asynchronisme du côté client et serveur

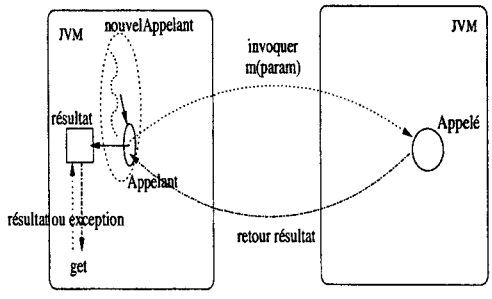



Figure 6.10 – Traitement des exceptions pour un appel asynchrone

Si le traitement à distance génère une exception, elle est récupérée à l'endroit où le traitement a été invoqué et remontée dans l'objet futur. L'objet futur retourné peut contenir, soit le résultat de l'appel, soit une éventuelle exception levée, pendant l'exécution, du côté serveur (voir la figure 6.10).

## 6.3 Conclusions

Ce chapitre présente la double expression du parallélisme en ADAJ, d'objets et de méthodes, simple et transparente pour l'utilisateur.

L'approche par collections d'objets distribués retenue en ADAJ permet une expression simple du parallélisme de données des applications. Des traitements parallèles multithreadés peuvent être lancés sur chaque fragment ; le parallélisme se retrouve aussi dans l'asynchronisme des opérations, les résultats étant récupérés à travers une structure spéciale qui a la fonctionnalité d'un objet futur. Les opérations attachées à ce type d'objets gèrent les exceptions utilisateur, éventuellement levées par l'exécution des traitements.

De point de vue fonctionnel, l'expression du parallélisme en ADAJ peut être mise en rapport avec celle de ProActive, projet mené concurremment. L'existence du concept d'objet actif en ProActive permet l'activation d'appels asynchrones, aspects réalisés de manière transparente en ADAJ (par une gestion cachée des threads), sur les objets remote de JavaParty. Les objets actifs et les objets remote ont des caractéristiques semblables : création à distance, possibilité de migration, transparence d'appel. Ils diffèrent dans la sémantique d'appels : les appels sur les objets actifs sont asynchrones, mais sur les objets remote ils sont synchrones. Le choix d'ADAJ de conserver le modèle d'objets remote de JavaParty est dû à la sémantique proche des objets distants RMI,

fournis par le modèle distribué Java.

Les groupes ont été construits ainsi de manière incrémentale, en ProActive, sur les objets actifs, afin d'exprimer le parallélisme de données. En ADAJ, le parallélisme de données est attaché à une collection distribuée, formée par un type particulier d'objet remote, le fragment. Leur construction respective est différente : le groupe contient des objets actifs précédemment créés, tandis que la collection distribuée crée elle-même des fragments vides, remplis ensuite par l'utilisateur. Ce comportement interdit en ADAJ le partage d'un fragment par plusieurs collections distribuées ; en ProActive, le partage des objets par différents groupes est admis. Cette sémantique facilite en ADAJ la cohérence des traitements multithreadés. La hiérarchie de collection distribuée permet une communication entre des fragments d'une même collection, fonctionnalité non exploitable en ProActive. La collection distribuée réunit un type spécifique de fragments[1], acceptant un polymorphisme de types, comme les groupes polymorphes de ProActive.

L'approche parallélisme de données en ADAJ, aussi comme en ProActive, diffère de l'expression parallèle proposée en Do !. Basé sur l'utilisation des frameworks, Do ! sépare la vue de données de celle de tâches, à l'aide du motif d'opérateur. Des tâches définies par l'utilisateur sont réunies dans une collection, qui sera associée à un constructeur parallèle. En ADAJ, les données et les tâches pouvant être appelées sur ces données sont regroupées, grâce aux concepts généraux, d'instance et de classe.

La gestion des retours des appels asynchrones est réalisée à travers un objet futur. Pour les communications de groupe, en ProActive, un autre groupe est renvoyé, mais en ADAJ, un collecteur est utilisé, qui a plus la fonctionnalité de contrôle des synchronisations, que de gestion des résultats.

L'entité de placement en ADAJ est le fragment : des distributions différentes (cyclique ou par bloc) sont proposées, indépendamment des caractéristiques de l'exécutif (notamment le nombre de machines). Une approche similaire, transparente, pour le placement des objets actifs, est visée en ProActive, mais une description de la correspondance entre les nœuds virtuels et les JVM est exigée. Ainsi, le déploiement en ProActive peut être repoussée au moment de la définition de l'exécutif. En Do !, le placement des éléments de la collection est contrôlé par un modèle de distribution attaché à la collection, sans une migration possible pendant l'exécution.

Dans la globalité du développement d'une application distribuée et parallèle, différentes approches se retrouvent :

- la description de l'application sous forme de programme parallèle, complété ensuite par des spécifications de distribution (projet Do !),
- la définition d'une sémantique asynchrone et distribuée, enrichie de types de communications particuliers (projet ProActive),
- l'expression d'un double parallélisme, dans le contexte d'une sémantique distribuée (projet ADAJ).

Développées en Java, les collections distribuées et les appels asynchrones sont introduits sans aucune modification de la machine virtuelle Java. La sémantique d'objet remote (héritée du modèle d'objets JavaParty), l'équivalent de l'objet distant RMI, permet une distribution implicite, ou explicite, une transparence d'appel, une facilité de migration.

Deux types d'implémentations sont fournis, à base de réflexion, ou de génération de code, présentés dans le chapitre suivant. La version utilisant l'introspection n'offre pas de typage pour les appels, tandis que la deuxième version récupère cette propriété du langage Java.

[1] le type des fragments peut être final, ce que ProActive interdit pour les objets actifs

# Chapitre 7

# Mise en œuvre d'une bibliothèque d'outils parallèles

## 7.1 Introduction

Dans ce chapitre nous présentons de quelle façon s'exprime le parallélisme dans l'environnement ADAJ. L'accent est mis surtout sur la mise en œuvre des appels distribute sur les collections distribuées, les appels asynchrones étant un cas particulier moins complexe.

L'algorithme appliqué pour la recherche de la méthode à invoquer dans les appels distribute ou asynchrones, mécanisme connu sous le nom de *réflexion*, est présenté (section 7.2). La complexité liée à la recherche de la méthode applicable (voir le problème de *résolution de surcharge*) est compensée par la transparence offerte, pour les appels parallèles, sur des données distribuées (section 7.3).

L'inconvénient le plus important d'une telle approche se trouve dans l'inexistence d'un typage fort (section 7.4), pour le nom ou les types des paramètres des appels parallèles. Pour pallier cet aspect, dû au mécanisme de réflexion utilisé, une autre solution est proposée (section 7.5), qui présente plusieurs avantages :

- offre un typage fort pour les appels parallèles,
- évite le surcoût de la réflexion,
- facilite l'écriture pour l'utilisateur.

Cette solution, à base de génération de code, sera comparée, en conclusion, avec la solution à base de réflexion, en fonction de leurs caractéristiques.

## 7.2 Réflexion et résolution de surcharge

### 7.2.1 La réflexion

Les méthodes invoquées par un appel distribute ou par un appel asynchrone sont définies, par l'utilisateur, après la définition de la bibliothèque réalisant ces appels. Pour réaliser l'invocation de ces méthodes définies tardivement, le mécanisme de *réflexion Java* (appelé aussi *introspection*) [Sune] est utilisé. La réflexion API Java permet de représenter des classes, interfaces ou objets dans la machine virtuelle. Les outils proposés permettent de :

- déterminer la classe d'un objet,
- obtenir des informations sur les modificateurs, attributs, méthodes, constructeurs d'une classe et de ses super-classes,
- trouver les constantes et les déclarations de méthodes d'une interface,

- créer une instance d'une classe dont le nom n'est connu qu'à l'exécution,
- obtenir et modifier la valeur d'un attribut, même si le nom d'attribut n'est connu qu'à l'exécution,
- invoquer une méthode sur un objet, même si la méthode n'est pas connue à la compilation,
- créer un nouveau tableau dont la taille et le type d'éléments ne sont connus qu'à l'exécution et modifier les composants du tableau par la suite.

Deux types d'applications nécessitent la réflexion. La première catégorie contient des applications qui nécessitent l'utilisation de méthodes et attributs publics de la classe, pendant l'exécution. Des exemples de cette catégorie sont les services Java Beans. Le deuxième type d'applications nécessite de découvrir et d'utiliser les membres déclarés d'une classe. Ces applications ont besoin d'un accès, pendant l'exécution, à l'implémentation de la classe, fournie par le fichier *.class*. Dans cette catégorie d'outils de développement, s'inscrivent également la bibliothèque des collections distribuées et l'outil d'appels asynchrones proposés par ADAJ.

La réflexion est utilisée en ADAJ pour obtenir le nom de la méthode à invoquer et les types de paramètres formels, parce que la méthode n'est pas connue à la compilation de la bibliothèque d'outils parallèles ADAJ. Les méthodes de la classe *java.lang.reflect.Method* permettent la récupération de ces informations. De plus, la méthode *invoke*, de la même classe, est utilisée pour invoquer la méthode cherchée sur l'objet cible.

### 7.2.2 La recherche des méthodes applicables

En Java, lors de la recherche de la méthode à appliquer, le concept de *méthode applicable* est utilisé. Une méthode est dite applicable par rapport à une invocation d'opération si, et seulement si, les deux conditions suivantes sont remplies :

- le nombre de paramètres de la méthode déclarée est égal au nombre d'arguments de l'opération invoquée,
- le type de chaque référence est un sous-type du paramètre correspondant. Chaque argument représenté par un type primitif Java doit avoir une classe équivalente compatible avec le type du paramètre correspondant.

En ADAJ, le choix de la méthode à appliquer ne peut se faire qu'à partir du nom d'une méthode, d'où la distinction de différentes méthodes à travers les types de paramètres donnés.

Le mécanisme de réflexion peut directement trouver la méthode à appliquer, en connaissant le nom de la méthode et les types de paramètres, si la méthode est définie dans la classe (à l'aide de la méthode *getDeclaredMethod* de la classe *java.lang.Class*). Pour retrouver une méthode définie dans la super-classe, le mécanisme est plus complexe et l'algorithme est décrit dans la figure 7.1.

Lors de la récupération des résultats, le collecteur, au moment de sa définition, n'a pas d'information sur le type exact des résultats. Etant le plus général possible, le type de données du collecteur est *Object*. La généricité de la récupération des résultats impose que le type des données renvoyées ne soit pas primitif.

### 7.2.3 La résolution de surcharge

En utilisant la réflexion, deux difficultés apparaissent lors de la considération d'une méthode héritée et du polymorphisme concernant les paramètres. Dans le premier cas, naturellement, Java permet l'invocation d'une méthode d'une super-classe sur un objet, instance d'une sous-classe.

Le choix de la méthode à appliquer soulève des problèmes lors de la considération des paramètres. La compatibilité entre les types des paramètres actuels et des paramètres formels est testée pour :

- les types primitifs (int, double, float, long, short, byte, char, boolean),

```
récupérer toutes les méthodes définies dans la classe
    ou dans toute super-classe ;
retrouver une ou plusieurs méthodes dont le nom
    coïncide avec le nom de la méthode à invoquer
    et le nombre de paramètres est le même ;
si (aucune méthode est trouvée) alors
   l'appel n'a pas été bien construit ;
sinon
   tantque (la méthode n'est pas trouvée)
            récupérer les types des paramètres ;
            si (les paramètres passés sont des
                    instances des types trouvés) alors
              la méthode a été trouvée ;
            sinon passer à la méthode suivante ;
            fsi
   fintantque
fsi
```

Figure 7.1 – La recherche de la méthode applicable

– les types références.

Les compatibilités pour les types primitifs sont celles prévues par la documentation Java [Sunh]. Pour les types références, si le type du paramètre actuel est dérivé du type du paramètre formel, alors le type du paramètre est compatible.

La recherche de la méthode à appliquer a pour coût, celui de la réflexion.

Une solution pour faciliter la recherche des méthodes héritées, gérer le polymorphisme et, en même temps, réaliser une vérification de types à la compilation, pour les appels construits, est présentée dans la section 7.5.

## 7.3 Description Java

Un double parallélisme, de données et de tâches, à travers les concepts de collection distribuée, et respectivement, d'appel asynchrone, est exprimé, en ADAJ, par une syntaxe particulière, décrite par la suite.

### 7.3.1 Appels distribute

Pour les fragments, présentés dans la section 6.1.1, la hiérarchie des classes est illustrée dans la figure 7.2 (les classes sont marquées en rectangles et les interfaces qu'elles implémentent en ellipses. Le lien d'héritage est en ligne continue, tandis que le lien d'implémentation des interfaces est en pointillé). Les fragments sont des instances de la classe *RemoteFragment.* Dans la bibliothèque, d'autres classes héritant de cette classe sont fournies : *VectorFragment* et *StackFragment* qui facilitent l'utilisation des fragments sous la forme de vecteur ou de pile. L'utilisateur peut définir son fragment propre, dont le type est *MyFragment* dans l'exemple, en étendant la classe *RemoteFragment.*

Une collection distribuée est construite en spécifiant le type de fragment qu'elle contient :

```
DistributedCollection colDistr = new
                DistributedCollection("MyFragment") ; .
```

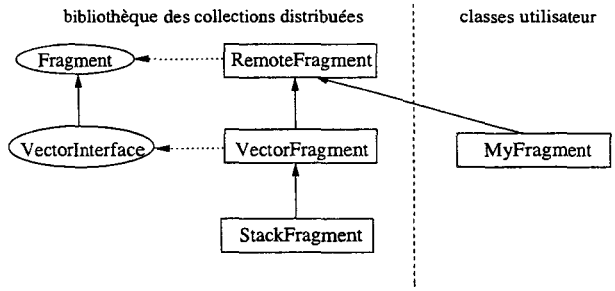



Figure 7.2 – Diagramme des classes de la bibliothèque et de l'utilisateur pour les fragments

La classe *MyFragment* désigne un fragment déclaré de la manière suivante :

```
class MyFragment extends RemoteFragment {
      public void methodVoid(...) { ... }
      public Object methodRes(...) { ...; return ...;}
}
```

Invoquer la méthode *methodRes* (qui retourne un résultat) sur tous les fragments de la collection distribuée *colDistr*, en parallèle, peut se réaliser à travers la méthode statique *distribute* (de la classe *DistributedTask*) de la manière suivante :

`Collector cRes = DistributedTask.distribute(colDistr, "methodRes", args) ;`

où `args` représente les paramètres de la méthode *methodRes*. Si la méthode ne prend pas de paramètres, alors l'invocation de la méthode *distribute* est :

`Collector cRes = DistributedTask.distribute(colDistr, "methodRes", null) ;`

Pour l'invocation de la méthode *methodVoid* (qui ne renvoie pas de résultat), la syntaxe est la suivante :

`Collector cVoid = DistributedTask.distributeV(colDistr, "methodVoid", args) ;`

où `args` est un tableau d'objets correspondant aux paramètres de la méthode.

Les différentes primitives distribute sont classées dans le tableau 7.1, en fonction du type d'appel (avec - *retour*, ou sans retour de résultats - *void*) et des paramètres pour la méthode appelée (les mêmes ou différents).

| type | paramètres différents | mêmes paramètres |
|---|---|---|
| void | *distributeVD* | *distributeV* |
| retour | *distributeD* | *distribute* |

Tableau 7.1 – Primitives distribute

Les arguments des primitives distribute et distributeV sont rangés dans un tableau unidimensionnel, et dans un tableau bidimensionnel (un tableau pour chaque fragment) pour les primitives distributeD et distributeVD.

Chaque élément du tableau unidimensionnel des paramètres correspond à un paramètre de la méthode pour un fragment, ce qui crée une correspondance bijective entre le nombre de fragments et la taille du tableau. En ProActive, les paramètres supplémentaires sont ignorés, mais en ADAJ, une vérification de leur nombre est réalisée. Dans le cas d'un paramètrage différent de la méthode activée par le distribute, pour chaque fragment, le paramètre `args[i]` du tableau bidimensionnel correspond aux arguments de la méthode s'exécutant sur le fragment d'indice `i`.

La récupération des résultats, ou l'attente synchrone de la fin de l'exécution de tous les traitements, est réalisée à travers des méthodes liées au collecteur :

```
// attendre un premier résultat
   PackRes res = cRes.getOne();
// récupération du résultat
   Object result = res.getResult();
// attendre la fin de tous les traitements
   cVoid.waitEnd();
```

### 7.3.2 Appels asynchrones

Un appel asynchrone s'effectue sur tout objet remote, y compris le fragment, par le biais d'une méthode statique de la classe *Asynchronous*. Pour un fragment *obj*, dont la classe *MyFragment* est définie précédemment, les appels asynchrones des méthodes déclarées se réalisent de la manière suivante :

`Return rV = Asynchronous.mVoid(obj, "methodVoid", args);`,
pour un appel asynchrone de la méthode *methodVoid* sur l'objet remote *obj*, ou

`Return rR = Aysnchronous.mReturn(obj, "methodRes", args);`,
pour un appel asynchrone de la méthode retournant des résultats, *methodRes*, sur l'objet remote *obj*.

Le type *Return* est le type d'objet futur pour les appels asynchrones, et la récupération des résultats s'effectue par la manipulation de l'objet *rV* (`rV.waitE()`) pour l'attente de la fin du traitement, ou de l'objet *rR* (`rR.get()`) pour la récupération du résultat.

### 7.3.3 Exceptions

Un appel asynchrone est réalisé par l'emballage de l'appel lui-même, dans un thread. Le nom de la méthode et les paramètres sont échangés entre l'appelant et le thread. Les résultats retournés, ou une éventuelle exception levée, sont récupérés par le thread d'exécution et transmis au collecteur.

L'invocation de méthode dans un thread est réalisée à l'aide de la réflexion (voir la section 7.2), en utilisant la méthode *invoke* (de la classe *java.lang.reflect.Method*). Cet appel permet de détecter si la méthode invoquée a levé une exception, à l'aide de la classe *java.lang.reflect.InvocationTargetException*, objet récupéré par l'appelant. Dans ce cas particulier, le traitement correspondant à l'exception consiste à remettre à jour, dans l'objet futur, un attribut spécifiant toute exception apparue pendant l'exécution de la méthode.

Dans la suite, tout accès au collecteur, après l'occurrence de l'exception, avertit l'utilisateur de l'événement produit, à travers une exception particulière, *DistributeException*. Cette exception transmet le message d'exception apparue pendant l'exécution d'une méthode. Pour les méthodes *getI* et *getOne*, une éventuelle exception produite lors du traitement sur le fragment correspondant sera retournée, tandis que, pour les méthodes *getAll* et *waitEnd*, tous les messages d'exception, éventuellement apparus sur tous les fragments, sont concaténés, dans l'ordre des fragments et renvoyés à l'appelant. Ainsi, l'appel sur un collecteur *cAll* pour la récupération de tous les résultats a la forme :

```
try {
     cAll.getAll();
} catch(DistributeException e) {
     System.out.println("Exception lors de distribute : " + e);
}
```

Le mécanisme est semblable pour le traitement des exceptions apparues lors d'un appel asynchrone.

## 7.4 Inconvénients de l'approche

Java est reconnu comme un langage sûr, grâce à sa propriété d'être fortement typé. Les appels de méthodes qui ne sont pas conformes à la signature de la méthode déclarée, sont détectés à la compilation.

Les appels de distribute et les appels asynchrones sont construits sans aucune vérification de types à la compilation.

Prenons, par exemple, la déclaration d'un fragment de type *MyFragment* de la manière suivante (où *A* est une classe utilisateur) :

```
class MyFragment extends RemoteFragment {
      public void methodVoid(...) { ... }
      public Object methodRes(A a) { ...; return ...;}
}
```

Un appel de type distribute pour la méthode *methodRes*, sur tous les fragments de la collection distribuée *colDistr*, est construit de la façon suivante :

```
Collector cRes = DistributedTask.distribute(colDistr, "methodRes",
                                            new Object[]{new A()}) ;
```

La méthode appliquée est identifiée par un nom et un tableau d'objets du type le plus générique possible (*java.lang.Object*). Un appel mal construit (mauvais nom de méthode, mauvais nombre de paramètres ou mauvais type de paramètres) ne sera pas détecté à la compilation. Dans l'exemple, un appel de distribute construit avec les paramètres d'un autre type que *A*, compatible ou non, est accepté par le compilateur, même si, en réalité, il s'agit d'une mauvaise manipulation de types. Une vérification des types est réalisée à l'exécution, dans le mécanisme de réflexion, mais elle risque d'être tardive, pour des appels incorrects.

Un contrôle de types ne serait possible que si le distribute est construit après la déclaration des méthodes appliquées, ce qui n'est pas le cas. Etant générique, la bibliothèque des collections distribuées et d'appels asynchrones ne peut pas prévoir toutes les définitions de classes utilisateur. La solution trouvée, pour invoquer des méthodes déclarées par le programmeur, fait usage de la réflexion mais fait perdre le typage.

Le même procédé est utilisé pour la communication asynchrone (sur les objets actifs ou de groupe) dans ProActive, mais il conserve le typage, grâce au polymorphisme entre les objets standard (locaux) et les objets actifs. Ainsi, les objets actifs ont le même type que les objets locaux, ce qui rend une transparence syntaxique totale de l'appel. En ADAJ, une transparence syntaxique partielle est offerte, qui permet à l'utilisateur de connaître la sémantique des objets manipulés. L'inconvénient apparu est la perte du contrôle sur les méthodes pouvant être appelées de manière asynchrone, due à la perte du typage.

L'objectif est de trouver une autre solution pour assurer le typage fort et conserver les propriétés de la solution précédente (résolution de surcharge, méthodes applicables).

## 7.5 Préservation du typage fort et nouvelle description Java

La solution du problème de typage fort, repose sur l'introduction d'un analyseur de code et la génération d'une nouvelle classe qui assure les bons types pour un appel parallèle. En schématisant,

le mécanisme consiste à détecter toutes les méthodes publiques déclarées dans une classe correspondant à un fragment ou à un objet remote, et à générer une nouvelle classe qui contient les mêmes méthodes, dont le code est légèrement transformé.

### 7.5.1 Description générale

Plus exactement, pour la déclaration d'un fragment, dont le nom de la classe est *RClass*, la génération est montrée dans la figure 7.3.

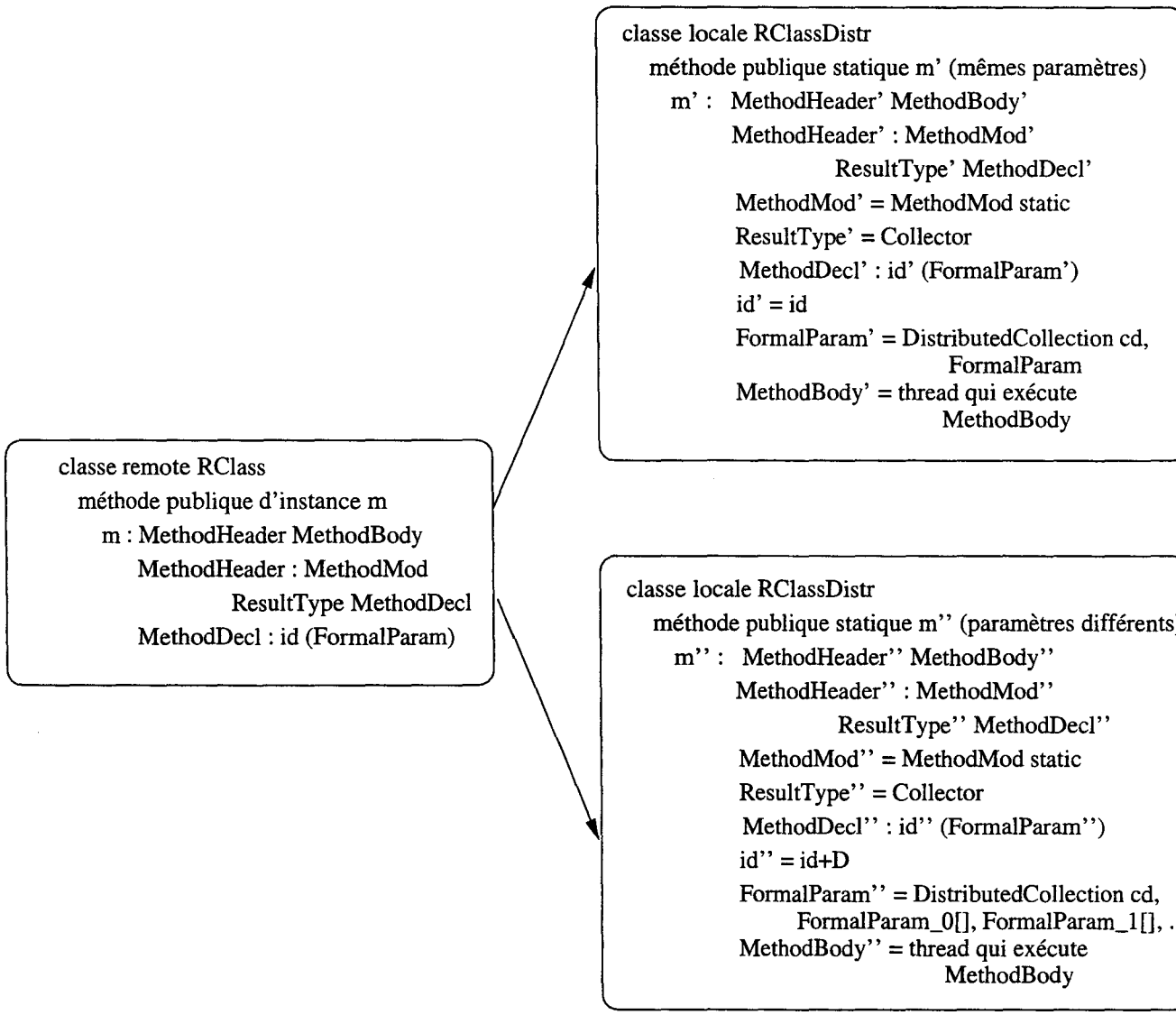



Figure 7.3 – Génération des classes

Toute méthode d'instance publique d'une classe remote, peut être accessible à distance par le biais d'une instance de la classe correspondante. Ainsi, la génération concerne exclusivement ces méthodes. Dans la nouvelle classe locale générée, à la méthode initiale correspond deux méthodes

statiques : une pour les appels du type distribute dont les paramètres sont les mêmes (figure 7.3 en haut à droite), et une pour les appels du type distribute dont les paramètres diffèrent d'un fragment à l'autre (figure 7.3 en bas à droite).

Dans le premier cas, le nom de la nouvelle méthode coïncide avec celle de la méthode initiale ($id = id'$). Le type du résultat retourné ($ResultType'$) est *Collector* qui accueille le vrai résultat, et les paramètres ($FormalParam'$) sont identiques aux paramètres de la méthode initiale, sauf pour la référence d'une collection distribuée qui est un paramètre supplémentaire.

Le corps de la méthode ($MethodBody'$) exécute la création d'un thread par fragment (récupéré du premier paramètre de la méthode représentant la collection distribuée). Chaque thread lance le traitement désigné par la méthode et les paramètres. Fonctionnellement, le code correspond au code de la première solution adaptée aux appels du type distribute.

Dans le deuxième cas, pour les méthodes prenant des paramètres différents pour chaque fragment, le nom de la méthode prend un $D$ ($id'' = id+D$), pour des raisons présentées dans la suite. Les paramètres de la nouvelle méthode générée sont : la collection distribuée et des tableaux, un pour chaque paramètre de la méthode initiale $m$, chaque élément d'un tableau ayant le même type, identique au type du paramètre de la méthode $m$ ($FormalParam = FormalParam\_0, FormalParam\_1$, ...). Les autres transformations sont semblables au cas précédent. Dans l'annexe A, nous montrons la génération, comparable, obtenue pour des appels asynchrones.

Ainsi, deux autres classes sont générées pour tout fragment (*MyFragment* dans l'exemple de la figure 7.4).

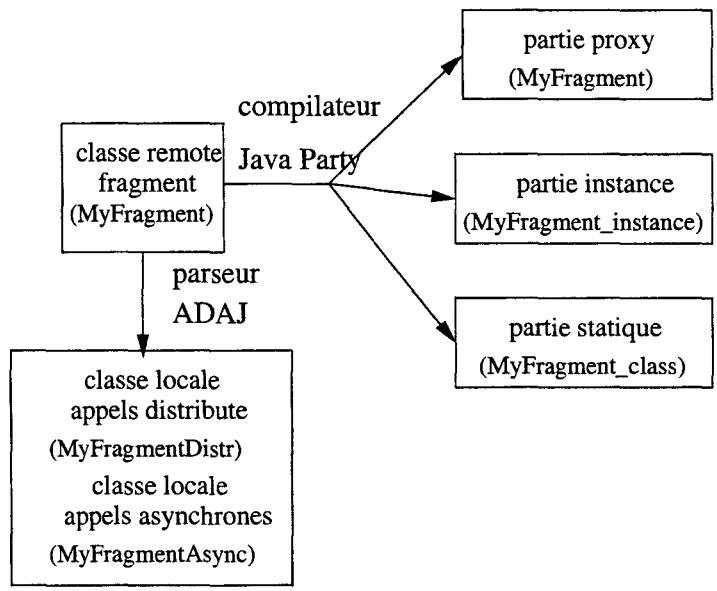



Figure 7.4 – Génération complète des classes en ADAJ

Pour une classe de type fragment (*MyFragment*), définie par l'utilisateur, qui contient 2 méthodes (voir la partie gauche de la figure 7.5), la génération de code crée une nouvelle classe (*MyFragmentDistr*), contenant 4 nouvelles méthodes, dont les signatures sont montrées dans la partie droite de la figure 7.5.

En reprenant la déclaration de la classe de la section 7.4, la syntaxe d'un appel du type distribute ou asynchrone change, de manière à être la plus proche possible de l'appel classique. On obtient, ainsi :

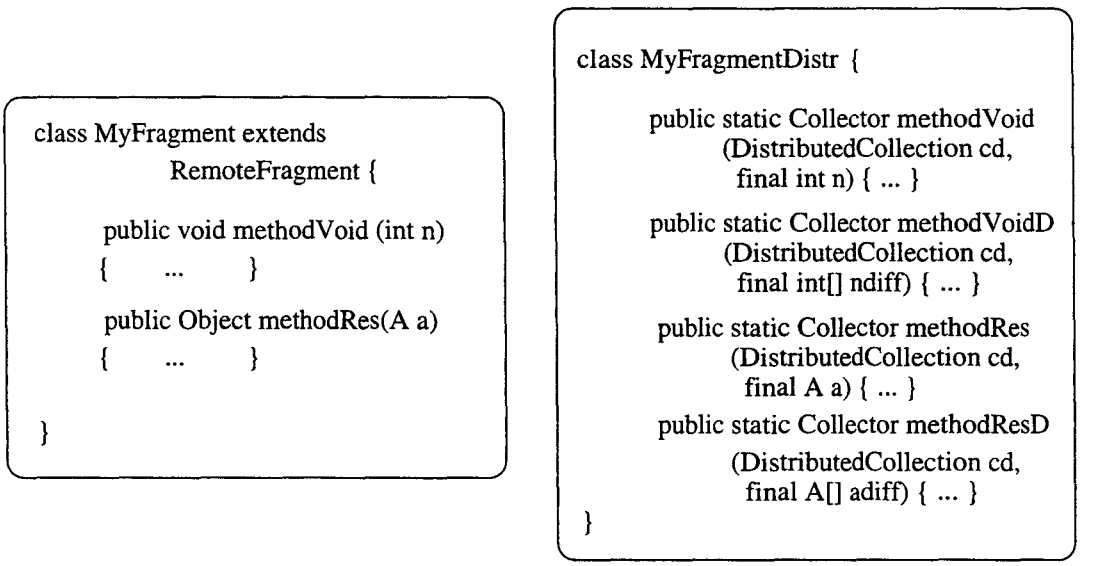



Figure 7.5 – Exemple de génération de code

```
Collector cRes = MyFragmentDistr.methodRes(colDistr, new A());
```
pour un appel du type distribute sur la collection distribuée *colDistr*, de la méthode *methodRes*, qui renvoie des résultats et prend un objet de type $A$ comme paramètre, et

```
Collector cRes = MyFragmentDistr.methodVoid(colDistr);
```
pour la méthode *methodVoid* qui ne renvoie pas de résultats et ne prend pas de paramètres.

Pour les appels du type distribute où les paramètres sont différents pour chaque fragment, au nom de la méthode est concaténée l'initiale $D$. Ainsi, l'invocation de la même méthode *methodRes* sur tous les fragments d'une collection distribuée, en passant des arguments différents pour chaque fragment, est faite de la manière suivante :

```
Collector cRes = MyFragmentDistr.methodResD(colDistr,
                 new A[]{new A(1), new A(2)});
```
Sans cette différence, il peut y avoir des conflits pour certaines déclarations de classes, comme dans l'exemple de la figure 7.6.

Le cas de conflit apparaît dans la situation où la classe initiale contient des méthodes surchargées dont les paramètres sont déjà étendus au tableau (un entier et un tableau d'entiers par exemple, comme dans la figure 7.6). Dans ce cas, la génération de code crée deux méthodes, ayant le même nom, l'une pour un lancement de distribute avec des paramètres différents, la méthode 2 dans la figure 7.6 qui a un tableau d'entiers comme paramètre, et l'autre pour un lancement de distribute avec des paramètres identiques, la méthode 3 dans la figure 7.6 qui a aussi un tableau d'entiers en paramètre. L'existence des deux méthodes ayant le même nom et la même signature, dans la même classe, génère une erreur de compilation en Java. En ajoutant l'initiale $D$ pour les méthodes appelables dans le distribute avec des paramètres différents pour chaque fragment, le conflit est éliminé.

La génération de la nouvelle classe est réalisée par un analyseur, écrit en JDK1.4, qui recherche des mots clés dans le code donné (*class*, *extends*, *implements*, *public*), en utilisant les expressions régulières ([Sunj]) et écrit du code correspondant aux transformations illustrées auparavant.

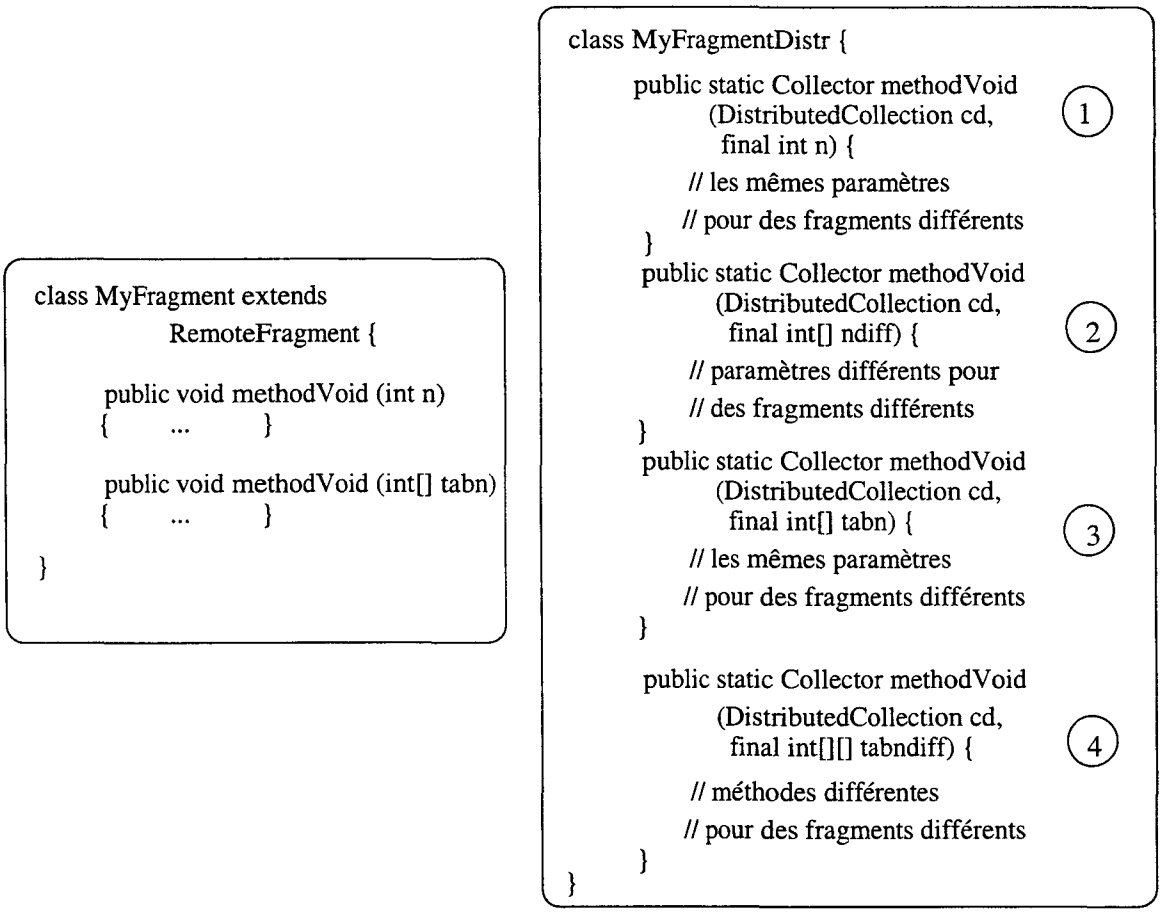



Figure 7.6 – Conflit de génération de code

### 7.5.2 Héritage

La deuxième solution de génération de code a l'avantage de retrouver le typage, mais le problème concernant le comportement de la génération de code, lors de l'héritage des fragments, est soulevé.

Pour les méthodes appelées dans un distribute, l'héritage devrait permettre les invocations des méthodes des super-classes fragment. Par exemple, pour une collection distribuée dont les fragments ont le type *MyFragmentH*, classe dérivée de la classe *MyFragment* (voir la figure 7.7), le distribute des méthodes déclarées dans la super-classe doit être valide. Le même comportement est exigé pour les appels asynchrones des méthodes des super-classes remote.

Pour les classes qui héritent des classes fragment, la génération de code conserve dans les classes locales générées, l'héritage qui existait entre les classes de départ. Dans le cas d'un appel du type distribute d'une méthode héritée, la méthode n'existera pas dans la classe directement générée depuis le fragment, mais elle sera héritée de la génération de la super-classe. Le mécanisme d'héritage de Java facilite la récupération de la méthode à appliquer. Le mécanisme de réflexion étant évité, la complexité de la recherche de la méthode présentée dans la section 7.2 est éliminée.

Un exemple partiel est présenté dans la figure 7.8, où la classe *MyFragmentH* hérite de la classe *MyFragment*. Le même lien d'héritage est conservé pour les classes générées, *MyFragmentDistr*, respectivement *MyFragmentHDistr*. Si une collection distribuée *colDistr* contient des fragments du type *MyFragmentH*, alors l'appel *Collector c = MyFragmentHDistr.mss(colDistr)* est valide et fait

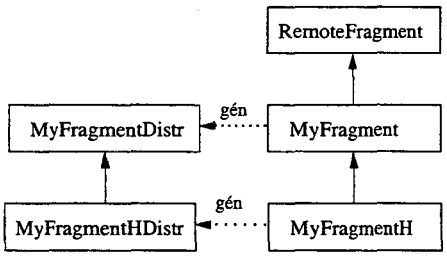



Figure 7.7 – Génération des classes pour des fragments "hérités"

appel à la méthode *mss* de la classe *MyFragment*.

*classes initiales*

```
class MyFragment extends RemoteFragment {
   public void mss()

   { ... }
}

class MyFragmentH
            extends MyFragment {
   public void mssH()

   { ... }
}
```

*classes générées*

```
class MyFragmentDistr {
   public static Collector mss(
      DistributedCollection colDistr)
   { ... }
}

class MyFragmentHDistr
            extends MyFragmentDistr {
   public static Collector mssH(
      DistributedCollection colDistr) {
   { ... }
}
```

Figure 7.8 – Exemple de génération des classes pour des fragments "hérités"

### 7.5.3 Résolution de surcharge

La résolution de surcharge consiste à choisir la déclaration de méthode à utiliser ([Sund]). En utilisant l'introspection, la résolution de surcharge était complexe à mettre en œuvre. La différence des types, entre les paramètres actuels et ceux formels, nécessitait la considération de toutes les méthodes de même nom et de même nombre de paramètres, la vérification des types pour la compatibilité et le choix de la méthode la plus spécifique.

Par la génération de code, le mécanisme Java standard de résolution de surcharge, est invoqué, évitant ainsi le surcoût de réflexion.

Dans l'exemple de la figure 7.9, *Interf* est une interface que la classe *InterfImpl* implémente.

En fonction du type de l'objet *param*, le choix de la méthode à appliquer sera fait : si le type déclaré de l'objet *param* est *Interf*, la méthode du fragment prenant un objet du type *Interf* comme paramètre, sera appelée, et si le type déclaré de l'objet *param* est *InterfImpl*, alors la méthode prenant un objet du type *InterfImpl* comme paramètre sera appelée. Le choix de méthode est fait par le mécanisme classique de résolution de surcharge de Java.

*classe d'un fragment*

```
public class DFragRes extends RemoteFragment {

    public void mss(Interf intf)
    { ... }

    public void mss(InterfImpl intf)
    { ... }
}
```

*appel distribute*

```
...
DistributedCollection colDistr =
    new DistributedCollection("DFragRes") ;
...
Interf param = new InterfImpl() ;
DFragResDistr.mss(colDistr, param).waitEnd() ;
...
```

Figure 7.9 – Exemple de résolution de surcharge

### 7.5.4 Traitement des exceptions

Le traitement des exceptions, levées pendant l'exécution d'une méthode, appelée de manière asynchrone, ou dans un distribute, est réalisé de la même manière que dans le cas de la réflexion. Les exceptions de l'application elle-même sont gérées à l'intérieur du thread de transition qui gère l'appel distant.

*classe initiale*

```
public class DFragExc
      extends RemoteFragment {

   public void affiche() throws Exception {

      ...

   }
}
```

*classe générée*

```
public class DFragExcDistr {

   public static affiche (
            DistributedCollection colDistr) {
     class ThreadAux extends Thread {
      Collector r ; ...
      public void run() {
       try {
        fragm.affiche() ;
       } catch (Exception e) { r...}
      }
     ...}
   }
}
```

Figure 7.10 – Traitement des exceptions dans la génération de code

L'exemple de la figure 7.10 montre la génération de code pour une classe qui contient une méthode qui lève une exception. Le code généré de la classe contient une nouvelle méthode qui gère la création et le lancement d'un thread de transition. Le vrai appel distant est encadré dans un bloc *try/catch* qui capte des exceptions de l'application, qui sont reportées dans le collecteur.

### 7.5.5 Problème non-résolu

Il n'y a pas de contrôle précis sur la corrélation entre le type des fragments d'une collection distribuée et les méthodes appliquées. Un cas particulier, présenté ci-dessous, le montre.

Considérons, par exemple (voir la figure 7.11), deux collections distribuées, *colDistr1*, *colDistr2*, qui contiennent respectivement les fragments de type *Frag1* et *Frag2*. Les deux fragments déclarent une même méthode *m*, ayant la même signature.

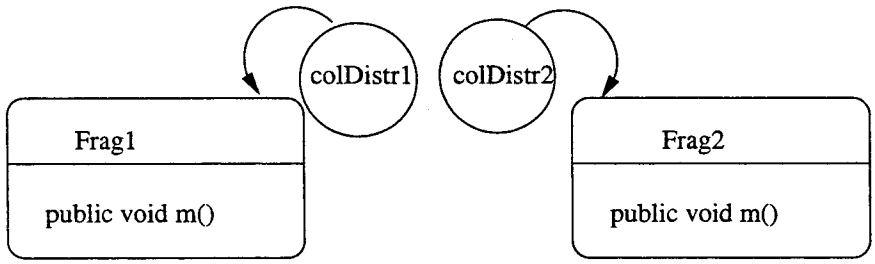



Figure 7.11 – Problème

Un appel du type distribute de la méthode *m* peut être invoqué sur les deux collections, l'appel *Collector c = Frag1Distr.m(colDistr1)* étant ainsi correct. Au contraire, l'appel *Collector c = Frag2Distr.m(colDistr1)* ne l'est pas et ne peut pas être détecté à la compilation, parce que l'identification, entre la classe des fragments et le type de fragments que la collection contient, ne peut pas être faite.

## 7.6 Conclusions

Dans ce chapitre, nous avons présenté de quelle manière est implémentée l'expression du parallélisme en ADAJ. Les fonctionnalités attachées à une collection distribuée (appels du type distribute, récupération des résultats, gestion des erreurs) et à tout objet remote (appels asynchrones) soulèvent quelques problèmes lors de l'implémentation.

Deux approches différentes ont été proposées et implémentées pour la mise en œuvre de l'expression du parallélisme en ADAJ. Une première version utilise la réflexion pour la transparence des appels parallèles sur des données distribuées. Vu l'inconvénient majeur du manque de typage, une deuxième version, qui utilise la génération de code, a été proposée. L'efficacité lors de l'exécution des deux méthodes est comparable, même si la méthode utilisant la réflexion a un très léger surcoût. La comparaison entre les deux méthodes est réalisée dans le tableau 7.2.

Une analyse plus fine des performances lors de l'utilisation d'une collection distribuée est réalisée dans le chapitre 9.

| *caractéristiques* | `version avec réflexion` | `version avec génération de code` |
|---|---|---|
| *typage* | non | oui |
| *résolution de surcharge* | oui (difficile) | oui (mécanisme Java) |
| *héritage* | oui (difficile) | oui (mécanisme Java) |
| *coût* | à l'exécution (de la réflexion) | à la compilation (de la génération de code) |

Tableau 7.2 – Comparaison des deux méthodes utilisées pour l'expression du parallélisme en ADAJ

# Chapitre 8

# Méthodologie de programmation

## 8.1 Outils pour la méthodologie de programmation

L'utilisation des collections distribuées et de l'asynchronisme en ADAJ induit une méthodologie de programmation parallèle de type MIMD, à l'opposé d'une approche de type BSP (Bulk Synchronous Programming) qui exige une contrainte de synchronisation à la fin de chaque étape de calcul.

ADAJ permet ainsi :
- d'exprimer aisément un parallélisme d'objets dans lequel les traitements sur les fragments sont activés de manière parallèle et asynchrone et qui autorise surtout une récupération totalement asynchrone des résultats,
- d'exprimer un parallélisme de méthodes,
- d'accroître naturellement la granularité des traitements en activant des méthodes sur les fragments (qui regroupent localement des objets) et non sur les objets individuels ou globalement dispersés,
- de ne pas nécessairement fixer lors de la conception du programme parallèle la granularité et le degré du parallélisme, mais de repousser ces choix au moment du lancement effectif du programme, en fonction du jeu réel des données à traiter et des caractéristiques du support d'exécution (en particulier le nombre de machines disponibles). Le degré de parallélisme peut même varier dynamiquement pendant l'exécution par ajouts et suppressions de fragments,
- d'être libéré des contraintes de localisation explicite des objets et de leurs migrations éventuelles.

Les exemples qui suivent illustrent l'analyse de plusieurs applications. Les différentes exécutions parallèles réalisées sont très satisfaisantes. Lors de l'analyse de ces applications, un accent particulier est mis sur la structure de données et, en vue de la distribution, sur le découpage réalisé. Les aspects détaillés sont présentés dans la suite, pour chaque problème.

Trois types d'applications ont été analysés :
- type académique, illustrant des problèmes de calcul matriciel (multiplication des matrices), de calcul vectoriel (manipulation des tableaux, calcul de la plus longue sous-suite),
- type optimisation combinatoire, par le problème du voyageur de commerce,
- type applications réelles, dans le domaine du datamining, par la recherche des règles d'association.

## 8.2 Manipulation des tableaux

L'utilisation des collections distribuées pour manipuler des vecteurs apparaît comme une pratique classique lors d'un calcul parallèle et distribué. Des configurations fréquentes se rencontrent dans la gestion des vecteurs, comme la simple addition. Les deux schémas de la figure 8.1 montrent deux configurations de cette addition de deux vecteurs : la première, élément par élément, en prenant les indices dans l'ordre, la seconde, position par position mais avec un décalage des indices. La gestion des résultats n'est pas traitée ici, étant similaire au cas de la multiplication des matrices, détaillé dans la section suivante.

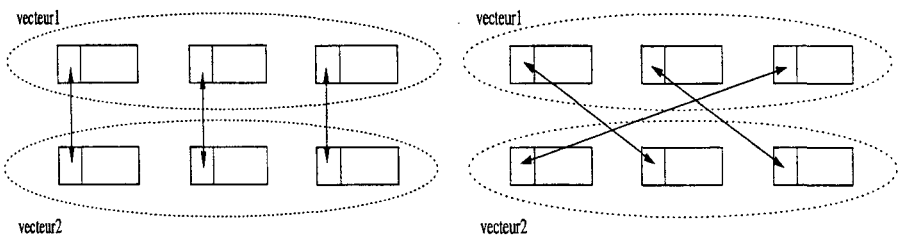



Figure 8.1 – Sommation de deux vecteurs

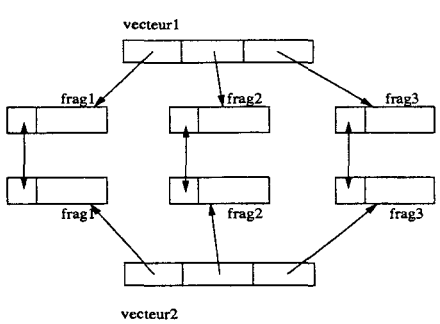
### 8.2.1 Sommation de deux vecteurs, élément par élément

Un vecteur, en ADAJ, peut être implémenté comme une collection distribuée (suite de fragments). Quand il s'agit de sommer deux vecteurs, l'opération considère deux vecteurs et renvoie un vecteur. En ADAJ, cette opération peut considérer :

- soit deux collections distribuées séparées,
- soit une seule collection distribuée qui contient deux vecteurs.

Le schéma des collections distribuées et la modélisation en ADAJ, pour les deux situations, sont illustrés dans les figures 8.2 et 8.3.

Dans le premier cas (figure 8.2), considérer l'équivalence entre une collection distribuée et un vecteur donne une flexibilité sur l'utilisation de plusieurs vecteurs, par une extension facile. En contre partie, rien n'assure que les fragments à additionner seront situés dans la même machine et les appels étant distants, on perd en performance lors des accès aux éléments du tableau. Dans

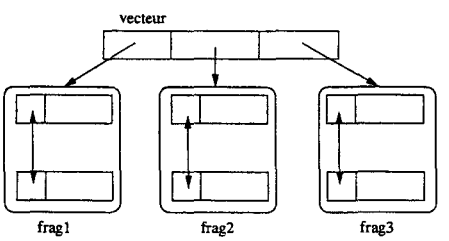



*classe fragment du sous-vecteur modélisé en ADAJ*

```
class FragMat extends RemoteFragment {
      int[] v;
}

DistributedCollection vecteur1 = new
        DistributedCollection("FragMat");
DistributedCollection vecteur2 = new
        DistributedCollection("FragMat");
```

Figure 8.2 – Sommation de vecteurs avec deux collections distribuées




*classe fragment du sous-vecteur modélisé en ADAJ*

```
class FragMat extends RemoteFragment {
      int[] v1 ;
      int[] v2 ;
}

DistributedCollection vecteur = new
       DistributedCollection("FragMat") ;
```

Figure 8.3 – Sommation de vecteurs avec une collection distribuée

la deuxième version (figure 8.3), on peut considérer que les deux sous-vecteurs sont contenus dans un même fragment, car chaque traitement effectue la sommation de deux valeurs, qui seront ainsi disponibles localement. L'efficacité est garantie, mais l'extensibilité par rapport à d'autres types d'opérations (non binaires) n'est pas aussi évidente, car la définition initiale de la classe prévoit des structures de données pour une opération binaire.

Les appels correspondant à la somme en pseudo-code s'écrivent :

- pour le premier cas, où la collection distribuée, correspondant au deuxième vecteur, est passée en paramètre :

```
somme (DistributedCollection vecteur2) {
  pour (i de 1 à taille()) faire
       somme[i] =  v[i] + vecteur2.getF(index()).v[i]
  fpour
}
```

  La méthode *index* renvoie l'indice du fragment courant dans une collection distribuée et la méthode *getF* récupère le fragment d'indice donné.
- pour le deuxième cas, les fragments de la collection distribuée contiennent deux vecteurs, donc la méthode effectuant la somme ne prend pas de paramètres :

```
somme () {
  pour (i de 1 à taille()) faire
      somme[i] = v1[i] + v2[i]
  fpour
}
```

où la primitive *taille*, définie par le programmeur, renvoie la taille du sous-vecteur (ou d'un des sous-vecteurs) du fragment.

### 8.2.2 Sommation des deux vecteurs voisins, position par position

L'intérêt est de faire la somme, élément par élément, des deux vecteurs, mais les éléments n'ont pas les mêmes indices. Par exemple, simuler la somme $somm[i] = vecteur1[i] + vecteur2[i+1]$.

Les deux cas de représentation ADAJ de la section précédente se retrouvent. La méthode à appliquer est :

- pour le premier cas (deux collections distribuées regroupent des fragments qui codifient, chacune, un vecteur),

```
somme (DistributedCollection vecteur2) {
  pour (i de 1 à taille()) faire
```

```
            somme[i] = v[i] + vecteur2.getF(index()+1).v[i]
      fpour
    }
```

- pour le deuxième cas, où la méthode *next* récupère la référence du fragment suivant celui courant,

```
 somme () {
   pour (i de 1 à taille()) faire
        somme[i] = v1[i] + next().v2[i]
   fpour
 }
```

Ce type d'application met en évidence la possibilité offerte par la collection distribuée de rendre les fragments directement accessibles à l'intérieur d'un autre fragment (à travers un indice donné ou des propriété de précédence). L'accès est a priori bloquant, mais l'utilisation des appels asynchrones de la bibliothèque ADAJ peut contourner cet aspect synchrone.

## 8.3 Multiplication de matrices

La multiplication de matrices est une opération classique de calcul matriciel, qui pose de problèmes de découpage et distribution lorsque les matrices sont de grande taille. Pour simplifier le problème nous considérons la multiplication des deux matrices $A$ et $B$, réalisée de manière parallèle et distribuée. En supposant la matrice $A$ découpée en groupes de lignes, trois possibilités existent pour la matrice $B$ :

- la dupliquer sur toutes les machines,
- la partager (la matrice $B$ étant sur une seule machine),
- la découper (en colonnes) et la distribuer.

Et, donc, deux présentations sont possibles, pour la matrice $B$ : soit la fragmenter, soit la conserver en une pièce. Les calculs correspondant aux deux situations sont présentés dans la figure 8.4.

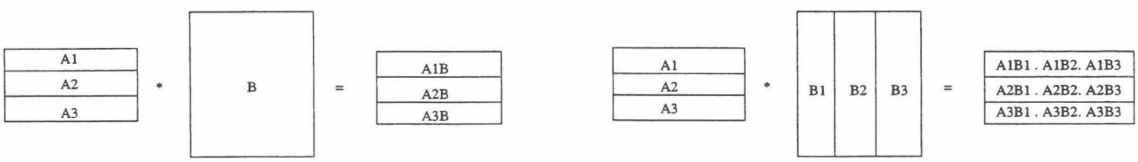



Figure 8.4 – Multiplication de matrices par duplication et par fragmentation

En ADAJ, les trois découpages de la matrice $B$ suggèrent trois solutions de structures possibles pour les deux matrices :

- A fragmenté, B fragmenté
  - ou deux collections distribuées, une avec les fragments contenant des lignes de $A$, l'autre avec des fragments des colonnes de $B$,
  - ou une collection distribuée, dont un fragment contient des lignes de $A$ et des colonnes de $B$,
- A fragmenté, B partagé,
  une collection distribuée dont un fragment contient des lignes de $A$,

- A fragmenté, B dupliqué,
  une collection distribuée dont un fragment contient des lignes de $A$ et la matrice $B$.

### 8.3.1 $A$ fragmenté, $B$ fragmenté

#### Cas 1 : 2 collections opérandes distribuées

La multiplication des deux matrices revient à un appel distribute sur la collection distribuée modélisant la matrice $A$, en lui passant comme paramètre la collection distribuée modélisant la matrice $B$.

```
class FragA extends RemoteFragment {
   // fragment de A
   SousMatrice fragA
   public void mult (DistributedCollection b) {
     pour (tout fragB dans b) faire
         // multiplication des sous-matrices
         mult (fragA, fragB)
     fpour
   }
}
```

Le résultat est une nouvelle collection distribuée, ayant le même découpage que la matrice $A$.

Cette solution offre de la flexibilité (car extensible à toute autre opération entre des matrices) et de l'homogénéité (les deux matrices, en entrée et la matrice résultante, sont représentées de la même manière). Contrairement à la conception facile, l'efficacité d'une telle solution est fortement dépendante de la granularité du traitement, tenant compte que tout accès à un fragment est un accès distant, donc coûteux.

#### Cas 2 : 1 seule collection distribuée

Le même découpage de A et B peut être modélisé autrement en ADAJ que par deux collections distribuées. Dans une autre solution, une seule collection distribuée contient des fragments ayant à la fois des parties de la matrice $A$ et de la matrice $B$. Le résultat peut être contenu directement dans le fragment ou à l'extérieur, comme le montre la figure 8.5.

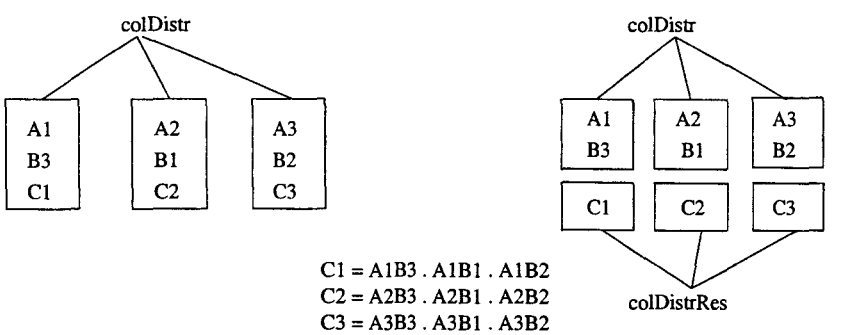



Figure 8.5 – Multiplication de matrices par fragmentation de la deuxième matrice

Le gain par rapport à la version précédente est la multiplication en local des parties de $A$ et $B$ situées dans un même fragment. L'inconvénient apparaît dans l'hétérogénéité des structures de

données : le résultat d'une multiplication de matrices est une matrice, mais la structure du résultat ne coïncide pas avec la structure de départ. Une multiplication ultérieure, en considérant que la matrice résultante a la même structure de la matrice $B$, doit mettre à jour les valeurs correspondant à la deuxième opérande dans la méthode de multiplication de matrices.

La structure proposée induit des opérations entre deux matrices et n'est pas extensible à des opérations n-aires.

### 8.3.2 A fragmenté, B partagé

La multiplication en parallèle des deux matrices revient à un appel distribute sur la collection modélisant la matrice $A$, en lui passant comme paramètre la structure de la matrice $B$. Etant partagée, la structure de la matrice $B$ est supposée distante.

```
class FragA extends RemoteFragment {
   SousMatrice fragA
   public void mult (B b) {
     fragA*B
   }
}
```

Le résultat est une nouvelle collection distribuée, avec le même découpage que la matrice $A$. Cette solution diffère des précédentes par la globalité de la matrice $B$, ce qui engendre une écriture facile mais moins efficace, de la multiplication (un appel à distance est compté pour chaque accès à un élément de la matrice $B$). En plus, tenant compte de la dimension de la matrice $B$, il peut y avoir des problèmes de stockage sur une seule machine.

### 8.3.3 A fragmenté, B dupliqué

La structure considérant la matrice $A$ fragmentée et la matrice $B$ dupliquée est une collection distribuée dont un fragment contient une partie de la matrice $A$ et toute la matrice $B$. Les éléments de la matrice B sont donc locaux au fragment, ainsi la multiplication est la plus efficace dans ce cas, grâce au nombre réduit d'appels distants, mais présente un inconvénient de mémorisation, lors d'une matrice $B$ de grande taille. Le résultat peut être récupéré comme dans le cas 2 de la section 8.3.1 dans le fragment lui-même ou dans une nouvelle collection distribuée (voir la figure 8.6). Cette solution présente le même inconvénient, d'être prévue pour des opérations binaires entre les matrices.

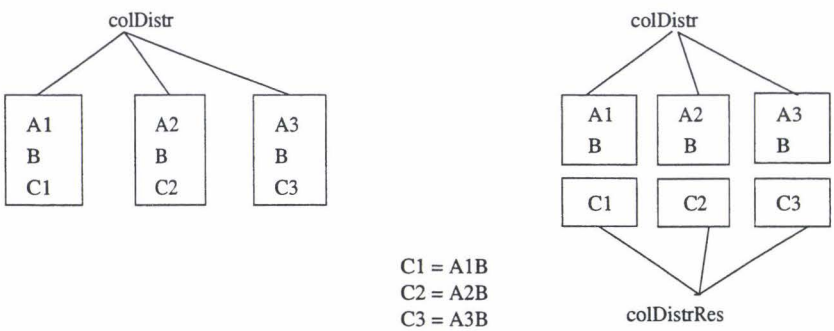



Figure 8.6 – Multiplication de matrices par duplication de la deuxième matrice

## 8.4 Problème de voyageur de commerce (TSP)

Les problèmes d'optimisation combinatoire utilisent des méthodes probabilistes pour trouver des solutions optimales à des situations trop complexes pour se prêter à une recherche exhaustive de toutes les possibilités. Ils peuvent donner lieu à un traitement parallèle. Un exemple bien connu est le problème de voyageur de commerce (TSP - Travelling Salesman Problem) qui doit visiter un certain nombre de villes en essayant de minimiser son trajet total. Parmi les méthodes couramment utilisées, nous nous sommes intéressés aux algorithmes génétiques [Dav91, Col99].

### 8.4.1 Principes des algorithmes génétiques et le problème TSP

Le schéma conceptuel d'un algorithme génétique appliqué à la résolution d'un problème comprend trois phases : la modélisation du problème (définir un environnement d'individus, appelé *population*), le cycle fini d'évolutions du système et l'interprétation de l'état final de la population, pour donner la solution au problème initial.

Pour le problème de voyageur de commerce, la population sera un ensemble d'individus, générés aléatoirement. Chaque individu représente une solution potentielle au problème posé.

Après avoir défini le codage de la population, *l'évolution* consiste à déterminer trois opérateurs : de sélection (sélection des individus appelés parents, les plus adaptés à être utilisés dans la prochaine population), de croisement (hybridation des chromosomes enfants à partir de ceux des parents), et de mutation (modifications des gènes qui apportent des innovations parmi les individus de la nouvelle population). Le schéma illustré dans la figure 8.7 présente un cycle d'évolutions d'une population d'individus :

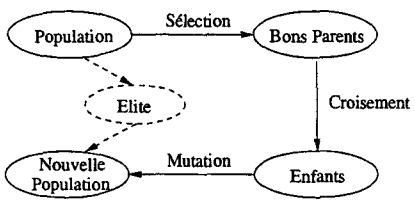



Figure 8.7 – Un cycle d'évolution d'une population

La figure montre également l'introduction possible d'une stratégie élitiste, qui assure que les meilleurs individus d'une génération seront automatiquement reproduits lors de la prochaine évolution.

Parmi les versions distribuées existantes, l'algorithme en îles [AK96, ZG99] suppose la fragmentation de la population initiale en sous-populations. Entre les sous-populations, il existe une propriété de voisinage, les sous-populations pouvant gérer des communications en anneau. Le processus itératif appliqué à la globalité de la population, *l'itération*, contient les étapes suivantes :

- lancer le processus de développement appliqué à chaque sous-population, caractérisé par un nombre fixe d'évolutions,
- attendre que tous les processus de développement soient finis,
- émigrer le meilleur individu de chaque sous-population vers la sous-population voisine (l'opération correspondante étant appelée *émigration*).

### 8.4.2 Modélisation du problème en ADAJ

Dans la suite, nous considérons un programme qui implémente un algorithme génétique générationnel[1] avec la stratégie élitiste, pour résoudre le problème de voyageur de commerce [Cah00].

La sémantique des objets remote en ADAJ est bien adaptée pour distribuer des populations sur le réseau, tandis que la collection distribuée peut profiter de l'aspect parallèle de l'application pour distribuer des traitements identiques sur différentes sous-populations.

La modélisation des données de l'algorithme génétique en termes de la collection distribuée est la suivante :

- un individu est une succession de toutes les villes à visiter (si $n$ est le nombre total de villes, un individu sera une permutation de $\{1, \cdots, n\}$), implémenté par la classe *Vector* de Java,
- chaque fragment est composé d'individus,
- la population, divisée en sous-populations, est implémentée comme une collection distribuée, dont les fragments sont des sous-populations.

Dans le tableau suivant nous montrons des transformations à apporter à la version séquentielle, pour obtenir sa version distribuée JavaParty[2] et respectivement la version distribuée ADAJ en utilisant les collections distribuées. Les modifications sont marquées en gras pour la version JavaParty, et en italique pour la version ADAJ.

*version séquentielle*

```
class Population extends Vector {
  public void traitement() {
      add(...);
  }
}
```

*version distribuée JavaParty*

```
remote class Population {
    Vector v;
    public void traitement() {
        v.add(...); // utilisation de v
    }
}
```

*version 1 ADAJ avec les collections distribuées*

```
class Population extends VectorFragment {
  public void traitement() {
      add(...);
  }
}
```

*version 2 ADAJ avec les collections distribuées*

```
class Population extends RemoteFragment {
  Vector v;
  public void traitement() {
      v.add(...); // utilisation de v
  }
}
```

A cause de la consommation d'héritage en JavaParty, un vecteur distant est modélisé par un attribut local de type vecteur d'une classe distante. Tout accès aux éléments du vecteur distant est redirigé à travers cet attribut. En ADAJ, deux possibilités existent :

- conserver le code séquentiel, en étendant la classe *VectorFragment*, qui est la modélisation distribuée d'un fragment particulier, de type vecteur (version 1 ADAJ),
- conserver le code JavaParty, en étendant simplement la classe *Population* de la classe *RemoteFragment* (version 2 ADAJ).

Les communications entre les sous-populations pour la version RMI se font de manière centralisée tandis que, pour la version ADAJ, elles peuvent facilement devenir distribuées grâce aux mécanismes offerts pour les collections distribuées, par des appels vers les fragments voisins.

[1] la taille d'une sous-population reste constante d'une évolution à l'autre

[2] les versions RMI et JavaParty étant similaires, seulement la dernière sera présentée

Les traitements sur les sous-populations dans la version RMI (ou JavaParty) ont la forme :

```
Population[] pop = ... ; // initialisation
for (i=0; i<n; i++)    // n est le nombre de sous-populations
    pop[i].traitement();
```

Dans le cas RMI, comme dans le cas JavaParty, pour éviter le synchronisme d'un appel distant, l'utilisateur doit mettre en œuvre des mécanismes qui traitent les appels asynchrones pour obtenir une meilleure efficacité.

```
class ThreadAsync extends Thread {
      Population pop;
      public ThreadAsync(Population pop) {
            this.pop = pop;
      }
      public void run() {
            pop.traitement();
      }
}
```

Le code parallèle inspiré de la version distribuée RMI (ou JavaParty) est le suivant :

```
Population[] pop = ... ; // initialisation
for (i=0; i<n; i++) {
    // création des tâches
    Thread task = new ThreadAsync(pop[i]);
    // lancement des tâches
    task.start();
}
```

En outre, des outils, non génériques, pour la collecte des résultats, doivent être prévus par l'utilisateur.

En utilisant les collections distribuées proposées par ADAJ, la construction des sous-populations et le lancement des traitements s'écrivent simplement selon la syntaxe suivante :

```
DistributedCollection colDistr = new
                    DistributedCollection ("Population");
Collector c = DistributedTask.distributeV (colDistr,"traitement",null);
```

Ce court exemple illustre la facilité d'expression d'un traitement parallèle et distribué à l'aide des collections distribuées de ADAJ (même sans aborder le mécanisme de récupération des résultats offert par le collecteur) par rapport à une version distribuée RMI ou JavaParty. De ce point de vue, ADAJ facilite donc grandement la conception de programmes parallèles et distribués.

## 8.5 Problème de la sous-suite

Une autre application qui montre l'utilisation des collections distribuées est l'algorithme qui calcule la longueur de la plus longue sous-suite croissante d'une série d'entiers. Les algorithmes distribués classiques considèrent la division de la série en plusieurs sous-séries, deux algorithmes distribués pouvant être implémentés :

- sans communications entre les sous-séries,
- avec communications entre les sous-séries.

L'exemple illustre particulièrement la facilité d'écriture en ADAJ de différentes versions de l'algorithme et la flexibilité offerte par l'usage des fragments.

### 8.5.1 Version sans communications

Le principe de l'algorithme distribué et parallèle, qui ne considère pas de communications entre les sous-séries, est le suivant :

- chaque sous-série calcule les valeurs locales : le premier élément de sa première sous-suite croissante et sa longueur, le dernier élément de la dernière sous-suite croissante et sa longueur, et la longueur de la plus longue sous-suite croissante,
- de manière centralisée, tous ces résultats sont récupérés, les bordures (les continuations d'une dernière sous-suite croissante d'une sous-série avec la première sous-suite croissante de la sous-série suivante) sont calculées dans un traitement global.

Le schéma de l'algorithme est présenté dans la figure 8.8.

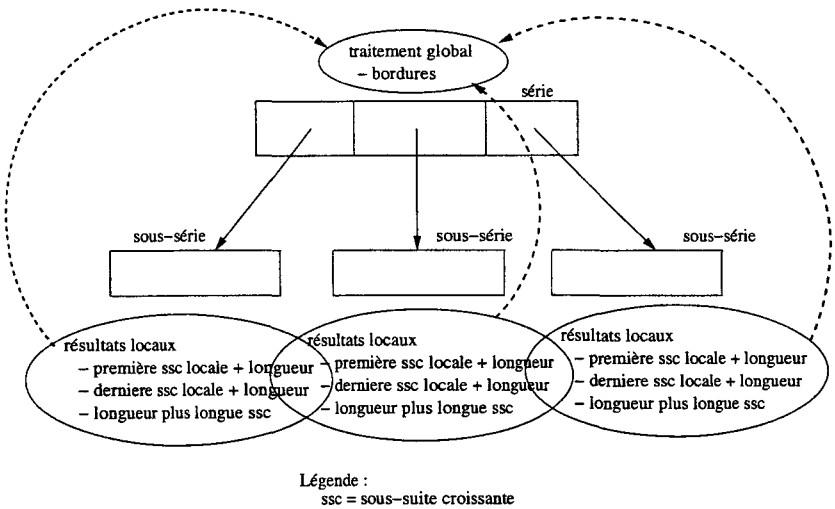



Figure 8.8 – Calcul distribué de la plus longue sous-suite croissante, sans communications

En ADAJ, une sous-série peut être modélisée par un fragment. Le fragment contient le codage de la sous-série (un vecteur, par exemple) et aussi des informations supplémentaires nécessaires au calcul local, contenues dans un tableau d'entiers, *Res* :

- le premier élément de la première sous-suite croissante locale (*premier*),
- la longueur de la première sous-suite croissante locale (*lgprem*),
- le dernier élément de la dernière sous-suite croissante locale (*dernier*),
- la longueur de la dernière sous-suite croissante locale (*lgder*),
- la longueur de la plus longue sous-suite croissante locale (*lgmax*).

Des méthodes appartenant au fragment effectuent la mise à jour de ces différentes données, la méthode appelée à distance renvoyant tous ces résultats. Le code ADAJ se trouve en annexe B.

Le fragment correspondant à une sous-série modélise l'utilisation d'un vecteur, mais en même temps, peut contenir des données exploitables par l'utilisateur, sans restreindre l'utilisation à un vecteur par exemple. Les méthodes qui manipulent les données peuvent également être définies sans aucune contrainte, de la même manière que pour toute définition de comportement pour une classe d'objets.

Le traitement global effectué crée et initialise la série comme une collection distribuée de sous-séries, lance le calcul en parallèle, récupère les résultats de tous les fragments et fait un traitement centralisé des bordures.

Le pseudo-code correspondant au traitement global est présenté dans la figure 8.9, où la primitive *taille* est définie par le programmeur précisant le nombre d'éléments d'une sous-série (le code ADAJ se trouve en annexe B).

```
main() {
  C : collection distribuée d'entiers;
  Res : collecteur de {premier, lgprem, dernier, lgder, lgmax};

  // calculs locaux parallèles sur les fragments
  Res = distribute (C, calcullocal);

  // traitement centralisé des résultats partiels
  Ri-1 = getI(Res,1);
  maxi = Ri-1.lgmax;
  pour i = 2 à nbfragments faire
       Ri = getI(Res,i);
       // traitement des bordures
       si Ri-1.dernier < Ri.premier alors
          // cas une seule sous-suite dans Ri-1
          si Ri-1.lgder = taille()-1 alors
             Ri.lgprem += Ri-1.lgprem
          sinon
             Ri.lgprem += Ri-1.lgder
          fsi
          Ri.lgmax = max (Ri.lgmax, Ri.lgprem);
       fsi
       maxi = max (maxi, Ri.lgmax);
       Ri-1 = Ri;
  fpour
}
```

Figure 8.9 – Psedo-code du traitement global de la plus longue sous-suite croissante

L'algorithme distribué et parallèle sans communications possède l'avantage de sa simplicité de conception et d'écriture pour le traitement local. Mais la complexité est reportée au traitement centralisé des bordures, qui suppose de considérer les éventuelles continuations d'une sous-suite croissante dans la sous-série suivante.

### 8.5.2 Version avec communications

La version distribuée et parallèle, avec communications entre les sous-séries, suppose un traitement global plus simple, mais aussi un traitement local plus complexe qui gère les communications avec des sous-séries voisines. Le principe de cet algorithme distingue deux phases :

- de calcul
  - chaque sous-série calcule sa dernière sous-suite croissante locale,

- chaque sous-série calcule la longueur de la plus longue sous-suite croissante locale,
- de manière centralisée, la dernière valeur est récupérée pour toute sous-série et la valeur maximale est calculée.

- de communication
  - chaque sous-série ayant une sous-série précédente, récupère de celle-ci son dernier élément et sa dernière sous-suite croissante calculée.

Le schéma de cet algorithme est présenté dans la figure 8.10.

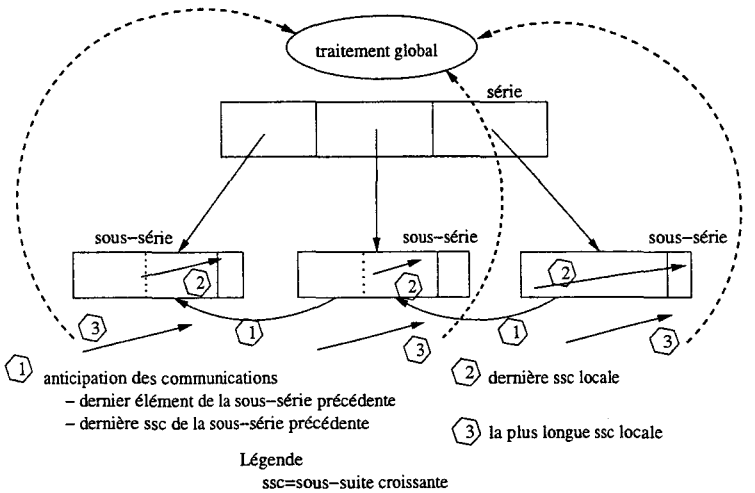



Figure 8.10 – Calcul distribué de la plus longue sous-suite croissante, avec communications

En ADAJ, la modélisation d'une sous-série est aussi réalisée à l'aide d'un fragment, dont les attributs sont, à part le vecteur codifiant la sous-série, les suivants :

- l'indice du début de la dernière sous-suite croissante (*debutdernier*),
- la longueur de la dernière sous-suite croissante (*lgder*),
- le dernier élément de la sous-suite précédente (*dernierprec*),
- la longueur de la dernière sous-suite croissante de la sous-série précédente (*lgderprec*).

L'anticipation des résultats nécessaires au calcul local est faite par un appel asynchrone, donc le type de ces deux derniers attributs est *Return*, le type de l'objet futur retourné par un appel asynchrone en ADAJ.

Le calcul de la longueur de la plus longue sous-suite croissante locale est le suivant (*verrou* étant un attribut utile pour optimiser le calcul de la longueur de la dernière sous-suite croissante) :

```
int maxlocal() {
  int lgcour = 1;

  // anticipation sur les communications
  dernierprec = asynch (fragmentprecedent, valeurderniere);
  lgderprec = asynch (fragmentprecedent, calcullgder);

  // calcul lgder si non déjà fait
  calcullgder();

  // calcul longueur maximale lgmax
```

```
  lgmax = lgder;
  si (debutdernier != 1) alors
     // plusieurs sous-suites croissantes dans le fragment
     i = 1;
     tant que (i < (taille() - lgder) faire
          si (tab[i] <= tab[i+1]) alors
             lgcour++; i++;
          sinon // rupture
             lgmax = max(lgmax, lgcour);
             lgcour = 1;
          fsi
     ftq
     si (il existe un fragment précédent) alors
        get (dernierprec);
        si (dernierprec < tab[1]) alors
           get (lgderprec);
           lgmax = max (lgmax, lgcour, lgderprec);
        fsi
     fsi
  fsi
  return (lgmax)
}
synchronized int calcullgder() {
  si (verrou = 0) alors
     // un appel à calcullgder() a été déjà fait,
     // donc le résultat est disponible
     return (lgder)
  sinon
    i = taille(); lgder = 1;
    tantque (i > 1 et tab[i] > tab[i-1]) faire
      lgder++; i--;
    ftq
    debutdernier = i;
    si (i = 1) alors
       // une seule sous-suite croissante dans le fragment
       get(dernierprec);
       si (dernierprec < tab[1]) alors
          get(lgderprec)
          lgder += lgderprec;
       fsi
    fsi
    verrou = 0;
    return (lgder)
  fsi
}
```



Le programme principal est considérablement moins complexe que dans le cas précédent, car il réalise seulement un calcul de maximum entre plusieurs entiers renvoyés par les calculs parallèles.

```
main() {
  C : collection distribuée d'entiers;
  Res : collecteur d'entiers;
  Res = distribute (C, maxlocal);
  pour (i de 1 à nbfragments) faire
       m = getOne (Res);
       maxi = max(maxi, m);
  fpour
  resultat = maxi;
}
```

L'algorithme distribué et parallèle, avec communications, effectue un traitement distribué des bordures. Une optimisation est facilement introduite qui fait le calcul de la dernière sous-suite croissante une seule fois. En même temps, le calcul global est considérablement simplifié.

## 8.6 Recherche des règles d'association en datamining

### 8.6.1 Présentation du problème

Le datamining est un terme général qui désigne, en réalité, un ensemble de traitements, différents, qui aboutissent à la découverte de connaissances variées, parmi lesquelles, la découverte de règles d'association. La recherche des règles d'association consiste à trouver des implications entre attributs des enregistrements d'une base de données. Ceci consiste à produire des règles de dépendances entre les attributs des relations, afin de prédire l'occurrence d'autres attributs. Ainsi, la notion *d'itemset*, comme ensemble d'attributs, est définie. Une mesure statistique, de *support*, est utilisée, qui permet de mesurer l'intérêt de l'itemset, ou sa fréquence dans le jeu de données. Cette information est enrichie avec la notion de *confiance*, qui permet de mesurer une réelle causalité entre la condition et la conclusion. La recherche des règles d'association revient à trouver les règles dont le support et la confiance sont supérieurs à certains seuils fixés au départ.

La résolution de ce problème, de la recherche de règles d'association dans une base de données, s'appuie sur des critères globaux de la base, vu que tous les attributs et tous les enregistrements sont analysés. La quantité de données importante à traiter impose le recours à la distribution et au parallélisme. Une solution déployable sur une grappe de PC doit assurer la conservation du critère global.

L'extraction des règles d'associations peut être décomposée en deux tâches :

- la recherche des itemsets fréquents (à la base de critères de support et de confiance),
- la génération des règles d'association à partir de ces itemsets fréquents.

En général, le traitement s'effectue sur une base hétérogène, d'où la nécessité de réaliser un pré-traitement de la base, pour la transformer en base binaire. Sur la base binaire, une partition sera réalisée, pour former des groupes d'enregistrements ayant des informations semblables. Les enregistrements les moins semblables seront placés dans des groupes différents. Cette phase de discrétisation, où des groupes sont formés, est appelée *clustering*. La recherche des itemsets fréquents peut se faire en parallèle sur des données distribuées. La distribution des données, correspondant à cette phase, est faite dans une phase de *distributing*, qui consiste à distribuer les données de manière à se rapprocher au maximum des critères globaux de la base de départ. Le traitement des données locales, pour effectuer la recherche des règles d'association, utilise l'algorithme Apriori [Zak99], dans une phase appelée *datamining*. Les différentes phases de traitement concernées dans la recherche des itemsets fréquents sont illustrées dans la figure 8.11 [Fio01]. La phase de clustering

traite la base fragmentée en colonnes. La base recompactée, issue du regroupement des colonnes, est divisée en lignes pour la phase de distributing. La phase de datamining traite également les lignes de la base transformée, en fonction des résultats issus de la phase précédente. Les résultats partiels de la recherche locale des itemsets fréquents sont regroupés pour déterminer les itemsets fréquents globalement.

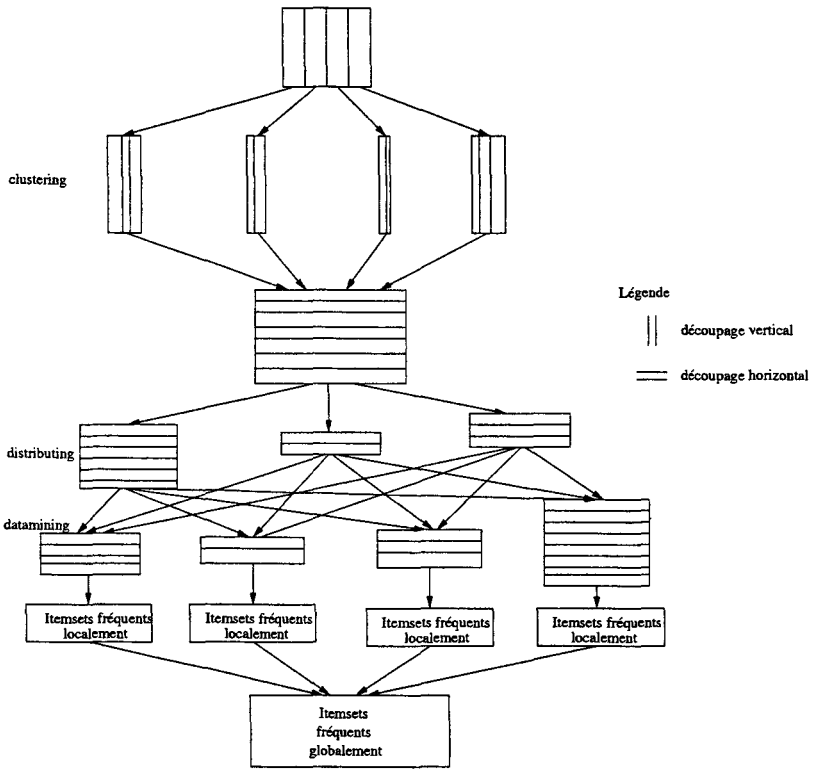



Figure 8.11 – Phases de traitement pour la recherche des itemsets fréquents

### 8.6.2 Modélisation ADAJ

L'environnement de programmation ADAJ offre différentes stratégies pour les architectures qui interviennent. L'exemple montre les différentes structures de données possibles en fonction de l'analyse du problème :

- une architecture totalement découpée,
- une architecture complètement groupée,
- une architecture fonctionnellement groupée.

### 8.6.3 Architecture totalement découpée

La première architecture envisagée consiste dans trois collections distribuées : une pour le clustering, une pour le distributing et une pour le datamining (figure 8.12). Chaque fragment clustering possède des références sur chaque fragment distributing. Les résultats issus du clustering sont envoyés directement aux fragments distributing. De même, les fragments distributing ont des références sur les objets datamining. L'appel de distribute sur la collection distribuée du clustering permet d'amorcer l'ensemble du traitement.

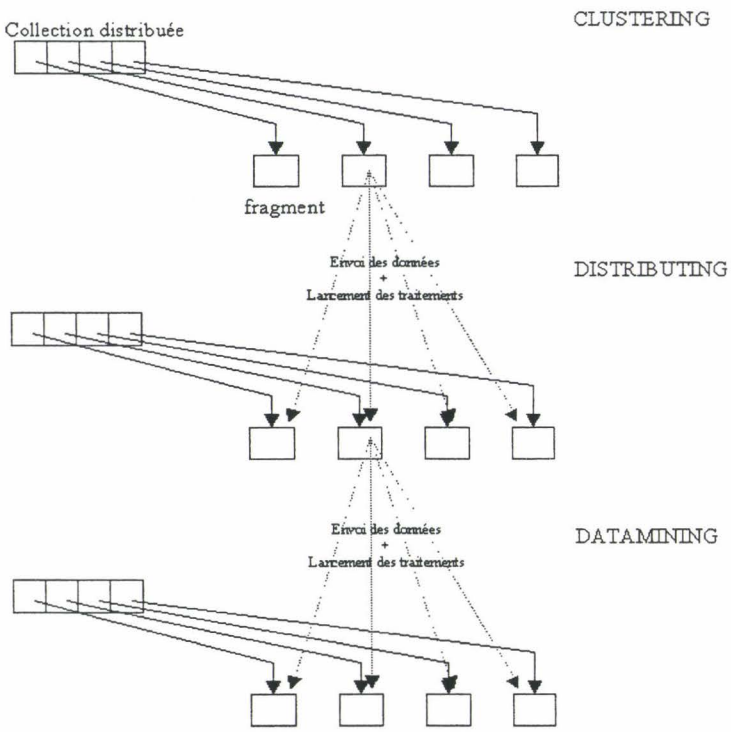



Figure 8.12 – Architecture totalement découpée

Cette solution ne profite pas des fonctionnalités attachées aux collections de fragments distributing et datamining.

### 8.6.4 Architecture complètement groupée

Cette architecture propose une seule collection distribuée dont les fragments gèrent les trois étapes, de clustering, de distributing et de datamining (figure 8.13).

La complexité de l'écriture des traitements repose sur l'utilisateur, car les nombreuses fonctions, s'exécutant de manière concurrente sur chaque fragment, doivent être synchronisées. Dans le cas particulier d'une base hétérogène, la phase de clustering s'impose pour homogénéiser les données. Supposons que cette phase soit réalisée au préalable, l'utilité de l'inclure, comme traitement correspondant au fragment, n'existe plus.

### 8.6.5 Architecture fonctionnellement groupée

Une autre architecture ressemble à la première, mais avec une gestion différente des appels de traitements (figure 8.14). Cette architecture est orientée par le type de traitements appliqués aux fragments. Etant une phase unique pour le pré-traitement de la base, la phase de clustering est indépendante et son résultat peut être sauvegardé dans un fichier (ou dans des fichiers distribués) pour être réutilisé, dès que l'opération sera reprise.

La phase de clustering est réalisée à part, sur une collection distribuée contenant des fragments clustering. Les résultats issus de cette phase initialisent les fragments de la collection distribuée distributing. Le distributing envoie ses données traitées aux fragments de la collection distribuée datamining.

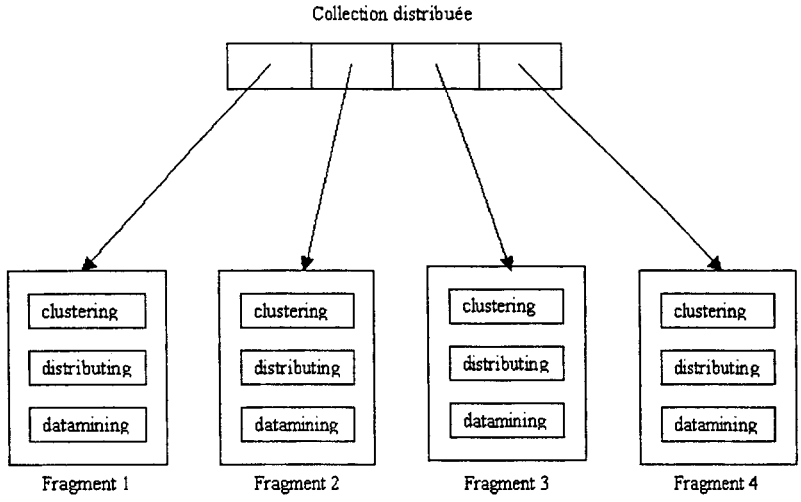



Figure 8.13 – Architecture totalement groupée

**Les synchronisations**

- Lors du clustering
  Pendant l'étape de clustering, des paquets sont envoyés aux fragments. Chaque fragment reçoit plusieurs paquets et les traite un par un. L'anticipation des calculs peut être faite en commençant le traitement, avant que tous les paquets soient reçus. Une file d'attente peut simuler le comportement correspondant.
- Lors du distributing
  Le même principe d'anticipation peut être utile, en commençant le traitement de distributing avant que tous les paquets ne soient arrivés.
- Lors du datamining
  Les fragments distributing envoient les données traitées aux fragments datamining. Mais le traitement ne peut commencer qu'au moment où tous les paquets ont été reçus, donc après que la phase distributing soit entièrement terminée. Ainsi, entre les deux distribute correspondant aux phases de distributing et datamining, il y a une phase de synchronisation totale.

## 8.7 Conclusions

Dans ce chapitre, différentes applications ont été présentées pour illustrer la méthodologie de programmation ADAJ. Grâce à l'utilisation des collections distribuées ou de l'asynchronisme, un modèle de programmation de type MIMD se distingue.

L'asynchronisme d'appel permet d'exprimer un parallélisme de méthode, illustré par l'application sous-suite, qui anticipe le calcul de certains résultats par des appels asynchrones sur les fragments voisins.

Un parallélisme d'objets est exprimé grâce à l'utilisation des collections distribuées, qui permettent d'activer des traitements parallèles sur les fragments. Le problème d'optimisation combinatoire, le voyageur de commerce, résolu à l'aide d'un algorithme génétique générationnel, est facilement modélisé, dans sa version distribuée de l'algorithme, à l'aide des collections distribuées.

Comme l'illustrent les exemples, la conception d'applications distribuées en ADAJ, en s'appuyant sur le concept de collections distribuées, réduit fortement l'effort du programmeur pour

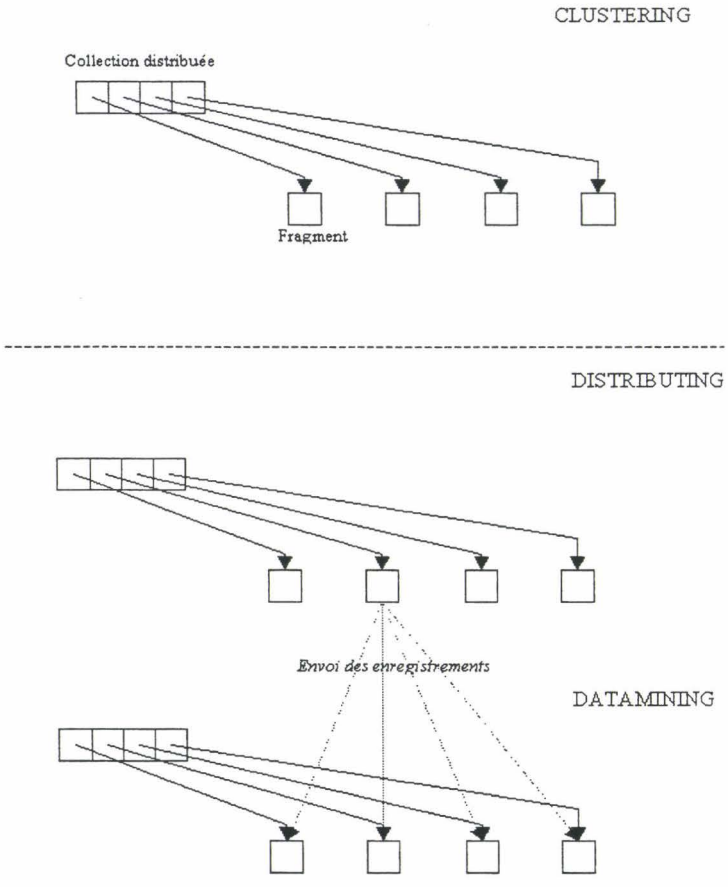



Figure 8.14 – Architecture fonctionnellement groupée

gérer le parallélisme.

Lors d'un parallélisme d'objets, le problème de la gestion du degré et de la granularité du parallélisme se pose. La granularité est accrue naturellement en ADAJ grâce aux découpages différents des matrices, par exemple, dans le calcul matriciel, ou grâce aux fragments contenant des quantités différentes de données, dans les différentes phases de calcul pour la recherche des règles d'association. Le degré du parallélisme est géré par l'ajout ou la suppression des fragments, dynamicité retrouvée dans l'application datamining. Les décisions sur le degré ou la granularité du parallélisme exprimés à l'aide de la collection distribuée, peuvent être repoussées à l'exécution. Au moment de la conception, le programmeur est libéré de les fixer, comme il est aussi libéré de toute décision de placement des fragments. Un programme parallèle en ADAJ s'appuyant sur les collections distribuées ou sur les appels asynchrones peut être conçu de manière tout à fait indépendante du support d'exécution et des paramètres effectifs, qui ne sont pris en compte effectivement qu'au lancement de l'exécution.

La facilité de la récupération des résultats est réalisée grâce au collecteur. La manipulation synchrone, ou asynchrone des résultats est démontrée par l'application du calcul de la plus longue sous-suite croissante. De plus, le problème de la recherche des règles d'association met en évidence une autre fonctionnalité du collecteur, celle de structure de contrôle, qui permet de gérer la synchronisation des traitements.

# Chapitre 9

# Evaluations

Le chapitre précédent illustre la facilité de conception et d'écriture des programmes dans l'environnement ADAJ. Grâce à une bibliothèque, le parallélisme est exprimé de manière transparente et la collecte des résultats est facilitée. La question qui se pose est : cette approche ne risque-t-elle pas de pénaliser l'exécution ? Pour y répondre, nous nous sommes intéressés d'abord aux performances obtenues lors de l'exécution parallèle, puis dans une deuxième étape, nous avons effectué une analyse comparative, plus fine, entre des programmes rédigés en ADAJ et en JavaParty.

L'application retenue pour cette évaluation est une version distribuée de l'algorithme génétique présenté dans le chapitre précédent pour la résolution du problème de voyageur de commerce. Le suivi et l'analyse des exécutions sont facilités par le caractère régulier et déterministe de cette application et par le choix d'un environnement homogène. Deux implémentations distribuées sont analysées, en JavaParty et en ADAJ, pour une même version parallèle, multithreadée, avec une même distribution des objets :

- la version JavaParty (notée *jp0*, évolution parallèle),
- la version ADAJ en utilisant les collections distribuées (notée *adaj0*, évolution parallèle),

où l'initialisation des sous-populations (la *construction*) est réalisée de manière séquentielle.

Les tailles des sous-populations considérées en distribué (une sous-population par JVM) sont de 1000, 1500 et 2000, le nombre d'itérations est de 1700 pour une instance TSP de 51 villes[1].

Les tests ont été effectués sur une grappe de 10 machines homogènes à processeur PIII à 733 MHz et 128 Mo de capacité mémoire, reliées d'un réseau à 100 Mb/s. Les programmes, dans la version 1.3 de JDK[2], ont tourné sur la plate-forme Linux 2.2.17 (Debian 2.2).

## 9.1 Accélération des applications ADAJ

Le gain de performance d'une application parallèle est mesuré par le quotient du temps qu'un seul processeur aurait mis pour exécuter le meilleur algorithme séquentiel, par le temps mis par l'ensemble des processeurs pour exécuter l'algorithme parallèle. Ce rapport de temps s'appelle *accélération*, notée $S$ (pour *speedup*).

Dans le calcul de l'accélération pour le programme analysé, deux remarques doivent être faites sur les algorithmes séquentiel et parallèle : l'algorithme séquentiel classique ne contient pas l'opération d'émigration, vu qu'il traite une seule population. Pour sa version distribuée, l'algorithme est légèrement transformé pour lier les sous-populations, par l'opération d'émigration. Les algorithmes

[1] le problème eil51.tsp
[2] Java Development Kit

qui résolvent des problèmes d'optimisation combinatoire, à base d'une heuristique, n'ont pas toujours la même solution d'une exécution à l'autre. Le cas d'arrêt considéré se base donc sur le nombre total d'itérations, dans le processus itératif d'évolution.

La taille de la population et le nombre d'itérations sont les deux paramètres à ajuster dans les versions séquentielle et distribuée. Diviser la taille de la population, en séquentiel, par le nombre de machines, pour la version distribuée, ou conserver la même taille de population en séquentiel pour chaque sous-population, en distribué, ne conduit pas à un même travail en séquentiel comme en parallèle. Il se pose le même problème pour le nombre d'itérations. Afin d'éviter le problème délicat lié à la comparaison des versions séquentielles et parallèles, distribuées, le programme séquentiel considéré est le modèle en îles séquentiel (sans l'utilisation des threads, pour éviter le surcoût de création des threads).

Théoriquement, l'accélération atteint la valeur maximale $M$, pour des versions parallèles, sans communications, quand elles s'exécutent sur $M$ machines. Pour le programme analysé, la figure 9.1 montre une accélération maximale de 1,95 (pour deux sous-populations de 1000 individus), avec la version JavaParty ou de 1,94 (pour la version ADAJ avec les collections distribuées). En augmentant la taille des données traitées, on remarque une croissance de l'accélération (par exemple, avec 4 sous-populations, on atteint des accélérations de 3,72 et 3,70 respectivement pour la version JavaParty et ADAJ avec les collections distribuées).

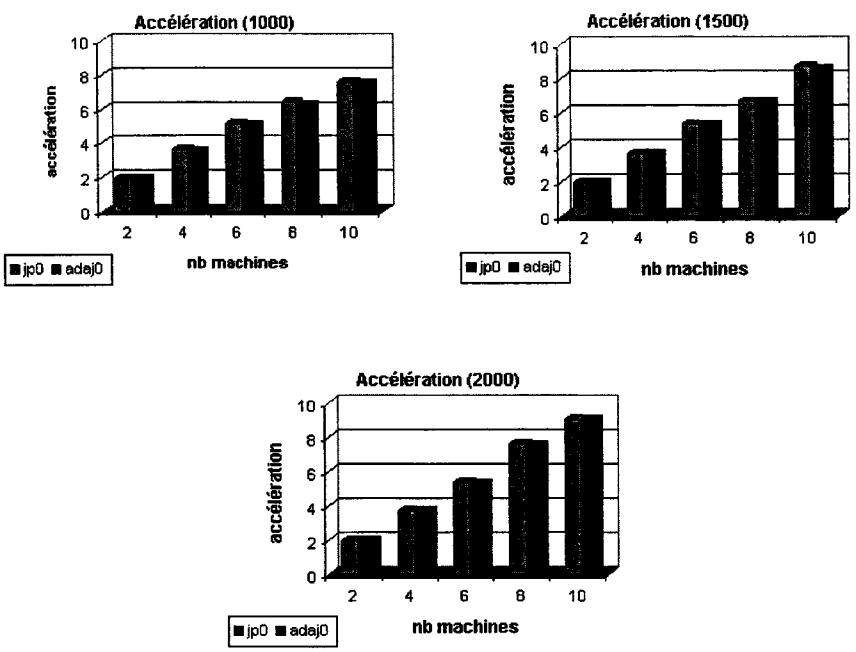



Figure 9.1 – Accélérations comparées JavaParty, ADAJ pour des sous-populations de tailles diverses

## 9.2 Surcoût des applications ADAJ

Bien que l'accélération acquise avec ADAJ s'approche de l'accélération obtenue avec JavaParty, la version ADAJ utilisant les collections distribuées est un peu plus lente. Il s'impose donc une analyse plus fine de la différence entre les temps d'exécution des deux versions.

### 9.2.1 Surcoût total

Le surcoût calculé est le pourcentage de la différence entre les temps d'exécution, des versions ADAJ et JavaParty, par rapport au temps d'exécution de la version JavaParty.

$$ST = \frac{ADAJ - JP}{JP}$$

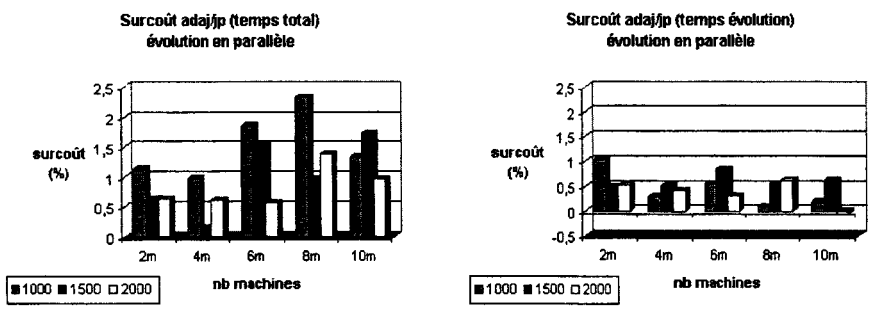



Figure 9.2 – Surcoût de l'utilisation des collections distribuées en ADAJ, cas 1 (évolution en parallèle)

La figure 9.2 présente le surcoût calculé par rapport aux temps totaux et aux temps de l'évolution, pour l'algorithme exécutant en parallèle l'évolution et pour des sous-populations de tailles 1000, 1500 et 2000. Le surcoût calculé (moyenne des trois exécutions, voir la partie gauche de la figure 9.2) montre une très légère différence de la version ADAJ par rapport à une version identique (mais explicite, dans la manière d'exprimer le parallélisme) en JavaParty. Une analyse plus fine montre que ce surcoût (qui atteint un maximum de 2,33 %) provient essentiellement de la construction initiale des sous-populations, réalisée de manière séquentielle : la partie droite de la figure 9.2 illustre un surcoût limité à l'exécution de la seconde phase (itérations stricto sensus). Dans ce cas, le surcoût n'atteint pas 1,05 %.

La figure 9.3 montre le surcoût de l'utilisation d'une collection distribuée, dans le cas où les phases de construction et d'évolution des sous-populations sont toutes deux effectuées de manière parallèle (en ADAJ et en JavaParty). Les sous-populations considérées ont les tailles 1000, 1500 et 2000 et le surcoût est calculé par rapport aux temps totaux et aux temps de l'évolution. Dans ce cas, le surcoût relatif aux temps totaux ne dépasse pas 1,01 %, à cause de la diminution de l'importance du temps de construction des sous-populations par rapport au temps total d'exécution.

Le surcoût de l'utilisation des collections distribuées est estimé, en moyenne pour l'application présentée, à 1,09 % (construction séquentielle et évolution parallèle) et à 0,56 % (construction et évolution parallèles). En ne considérant que les temps d'évolution, le surcoût constaté dans les deux cas se situe autour de 0,44 % (négligeable pour des temps d'exécution variant entre 15 minutes et 45 minutes). Les résultats montrent l'importance de la granularité du parallélisme. Les figures montrent aussi des surcoûts proches de 0 (voire négatifs) qui s'expliquent par des fluctuations dans le réseau utilisé, qui ne sont pas gérées par l'application ou l'environnement d'exécution.

En conclusion, le surcoût lié à l'utilisation des collections distribuées en ADAJ par rapport à JavaParty est donc proche de 0 (de l'ordre 0,5 %). Les collections distribuées n'apportent pas

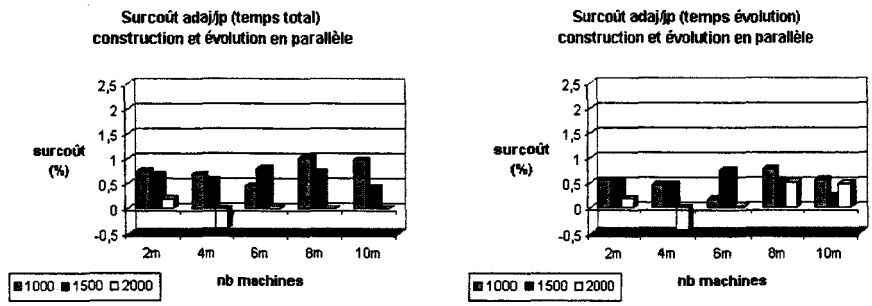



Figure 9.3 - Surcoût de l'utilisation des collections distribuées en ADAJ, cas 2 (construction et évolution en parallèle)

de surcoût important, et l'accélération obtenue est proche de celle d'une autre implémentation distribuée et parallèle.

### 9.2.2 Surcoût brut des communications ADAJ

Le surcoût total lié à l'utilisation des collections distribuées montre un léger ralentissement à l'exécution de la version ADAJ par rapport à la version JavaParty. Pour analyser plus finement l'origine du surcoût, des tests pour le calcul de surcoût brut d'une communication ont été réalisés. Ces tests ont porté sur deux types de réseau :

- hétérogène, formé par deux PC de caractéristiques différentes (un processeur PIII à 600 MHz, 128 Mo RAM, sous Linux 2.2.17 avec la version jdk1.3.0rc1 Java et l'autre, un processeur PIV à 1,66 GHz, 256 Mo RAM, sous Linux 2.2.2_2, avec la version 1.3.1 de Java), liés par un réseau de débit 10 Mb/s,
- homogène, formé par deux PC (processeur PIII à 733 MHz, 128 Mo RAM sous Linux 2.2.17, avec la version jdk1.3), liés par un réseau de débit 100 Mb/s.

Trois programmes de test ont été réalisés. Les traitements considérés, pour la classe d'un fragment, sont :

- une méthode de corps vide qui ne retourne rien (*Vide*),
- une méthode qui retourne un objet du type `Integer` (*Result*),
- une méthode qui exécute une boucle finie et retourne un objet du type `Integer` (*RB*).

Deux versions de programme, en JavaParty et en ADAJ, ont exécuté des appels successifs (10, 100, 1000 pour le réseau hétérogène et 10, 100, 1000 et 10000 pour le réseau homogène) de ces méthodes, pour un objet du type fragment, en JavaParty et pour une collection distribuée contenant un seul fragment, en ADAJ. De point de vue fonctionnel, les versions sont les mêmes, la différence est introduite au niveau de l'écriture et des bibliothèques utilisées.

Le surcoût calculé est, toujours, le pourcentage de la différence entre les temps d'exécution, des versions ADAJ et JavaParty, par rapport au temps d'exécution de la version JavaParty. Les résultats sont montrés dans les tableaux 9.1 et 9.2.

Le réseau hétérogène, de faible débit, masque les appels à distance, tandis que le réseau homogène, de grand débit, montre qu'un appel à distance vide, ou qui retourne un entier, est deux fois plus lent en ADAJ qu'en JavaParty, car effectivement il y a un appel à distance de plus qui est fait par la bibliothèque d'ADAJ (copie de la table des fragments). Dès que le traitement réalisé par

| surcoût adaj/jp (%) | *Vide* | *Result* | *RB* |
|---|---|---|---|
| 10 | 3,01 | 11,53 | 7,87 |
| 100 | 2,72 | 6,06 | -4,44 |
| 1000 | -4,92 | 3,32 | 0,68 |

Tableau 9.1 – Surcoût brut d'une communication ADAJ sur un réseau hétérogène

| surcoût adaj/jp (%) | *Vide* | *Result* | *RB* |
|---|---|---|---|
| 10 | 439,29 | 489,31 | 2,87 |
| 100 | 323,58 | 263,8 | -21,3 |
| 1000 | 205,01 | 196,59 | -30,8 |
| 10000 | 196,13 | 218,48 | -33,57 |

Tableau 9.2 – Surcoût brut d'une communication ADAJ sur un réseau homogène

la méthode devient non-négligeable (même de faible importance, voire environ 17 ms sur le réseau hétérogène ou 25 ms sur le réseau homogène), le surcoût diminue considérablement et la version ADAJ devient même plus efficace que la version JavaParty.

La différence importante signalée dans la décroissance du surcoût d'ADAJ, pour le nombre d'appels successifs 10, 100 et 1000, dans le réseau homogène, provient du temps très court de l'exécution d'une méthode vide ou d'une méthode qui retourne un entier (9/6,2/4,2 ms respectivement 15,44/9,07/6,09 ms en ADAJ par rapport à 1,68/1,48/1,4 ms et respectivement 2,62/2,49/2,05 ms en JavaParty). Une analyse comparative montre que, sur le réseau homogène, une méthode vide prend, en moyenne, 75,6 ms en ADAJ et 73,6 ms en JavaParty (pour 100 appels successifs). Ces résultats témoignent de l'importance de la granularité du traitement (donc la granularité du parallélisme) par rapport aux communications engendrées. Le traitement parallèle se trouve efficace quand ce rapport est supra-unitaire (supérieur à 1).

Le tableau complet des temps d'exécution pour chaque test se trouve dans l'annexe C.

## 9.3 Conclusions

Le chapitre précédent met en évidence la facilité d'expression du parallélisme et de récupération de résultats en ADAJ. Dans ce chapitre, nous nous sommes intéressés à ce que ces outils apportent en termes de performances d'exécution.

L'évaluation a porté sur la version distribuée de l'algorithme génétique pour la résolution du problème du voyageur de commerce. L'accélération obtenue pour une version ADAJ est comparable à celle d'un même algorithme implémenté en JavaParty. Des mesures plus fines, de surcoût total et de surcoût brut des communications, ont démontré l'importance du recouvrement des communications par les calculs.

La partie suivante de la thèse mettra en évidence les performances des collections distribuées mais dans le contexte d'un environnement d'exécution dans lequel se manifeste un mécanisme d'équilibrage de charge intra-application.

# Troisième partie

# Efficacité d'exécution des applications Java distribuées et parallèles

# Avant propos

La partie II montre l'intérêt d'une bibliothèque d'outils parallèles, pour la conception d'applications distribuées, conception censée rendre facile l'expression d'un double parallélisme, de méthodes et d'objets, et rendre efficace l'exécution.

Le deuxième volet d'obtention de l'efficacité consiste dans des mécanismes insérés au niveau de middleware qui permettent de redeployer l'application sur la grappe dans certaines situations (chapitre 10). Les projets cités dans le chapitre 11 présentent différentes approches possibles, soit à travers de redistribution d'objets, de placement de requêtes ou de placement initial. L'approche original en ADAJ pour résoudre ce problème est liée à l'introduction d'un mécanisme d'observation de l'application (chapitre 12) et à l'exploitation des informations issues de ce mécanisme pour en déduire une mesure de charge et un meilleur placement d'objets (chapitre 13). Les détails d'implémentation d'un tel environnement d'exécution sont présentés dans le chapitre 14, qui insiste notamment sur la technique de migration utilisée, une amélioration de celle proposée en JavaParty. Le mécanisme d'équilibrage de charge est validé par des expérimentations présentées dans le chapitre 15.

# Chapitre 10

# Problématique

## 10.1 Motivations

Les réseaux de stations de travail permettent d'augmenter la puissance de calcul disponible, ce qui rend possible le traitement des problèmes de plus grande dimension ou d'affiner la solution des problèmes de grandes tailles (comme la simulation des systèmes dynamiques, la résolution des équations différentielles, etc.). Ces machines constituent une plate-forme parallèle d'exécution pour différentes applications. L'analyse de la quantité de travail à exécuter sur ces plates-formes (ce qui sera définie sous le nom de *charge* dans le chapitre 12), pendant une période de temps, a montré la sous-utilisation des ressources, et donc la perte de potentiel de calcul disponible. L'exploitation efficace de la puissance de ces machines s'avère nécessaire.

Dans ce contexte d'exécution des applications distribuées sur des grappes de stations, s'impose l'introduction de mécanismes ou de politiques efficaces d'accès et d'utilisation des ressources (CPU, mémoire), par plusieurs utilisateurs qui veulent exécuter un programme. Les mécanismes de contrôle de ressources deviennent des intermédiaires entre les utilisateurs et les ressources demandées.

Certains environnements de programmation ou systèmes d'exploitation laissent le choix à l'utilisateur pour l'allocation des entités du programme, en lui offrant des primitives de placement [BAAD91] ou par l'intermédiaire d'une interface graphique [FBCL91]. Ces possibilités ne tiennent pas toujours compte du comportement dynamique de l'application ou de l'environnement d'exécution. Les algorithmes de distribution s'avèrent utiles pour automatiser et optimiser le placement. De plus, la transparence acquise par leur intégration dans la plate-forme, rend facile l'utilisation d'un tel environnement.

Différentes taxonomies des stratégies de distribution ont été proposées, qui permettent de classer les algorithmes en fonction de divers aspects. La taxonomie des algorithmes dynamiques de distribution proposée par Talbi [Tal97] s'appuie sur la considération de deux éléments de base : l'élément d'information et l'élément de contrôle. Le premier maintient l'information concernant l'état du système, information utilisée par le second pour réaliser la distribution. La classification des éléments d'information distingue deux types : *coopératifs* ou *aveugles*. Dans le cas coopératif, les éléments d'informations peuvent être *centralisés*, *hiérarchiques* ou *distribués*. D'autres caractéristiques sont considérées pour affiner la classification : *l'initiateur*, *la réactivité* aux changements dans le système, *la préemption* de l'exécution.

## 10.2 Modèles de distribution de charge

La taxonomie proposée par Casavant et Kuhl [CK88], pour classer les algorithmes de répartition de charge, différencie deux modèles, selon le mode de fonctionnement : statiques et dynamiques.

- *le modèle statique* est caractérisé par l'affectation des tâches aux processeurs de manière fixe, indépendamment de l'évolution de l'application ou des fluctuations dans l'environnement. Ce type de distribution affecte généralement les tâches de manière probabilistique ou heuristique, sans considérer des événements à l'exécution.
- *le modèle dynamique* propose une solution au problème d'apparition de charges non prévues, ou de variations pendant l'exécution, concernant la modification des demandes de ressources ou des disponibilités. La réactivité de l'algorithme face à ces variations est obtenue grâce à des mécanismes d'observation, qui permettent de capter et analyser des signaux susceptibles d'influencer le choix de la meilleure localisation et de la distribution correspondante. Ceci permettra à l'algorithme de s'adapter à la situation rencontrée (la charge inattendue).

Un troisième type [Fol92], *le modèle adaptatif*, variante du modèle dynamique de distribution, est caractérisé par une évolution dynamique des stratégies et des paramètres utilisés. En fonction du comportement de l'application, les paramètres sont ajustés afin d'éviter

- des actions inutiles dans le cas de charge uniformément distribuée et
- de réagir de manière trop prompte à des fluctuations de charge.

Par rapport à un modèle non-adaptatif, qui continue à redistribuer la charge, suivant la même politique, le modèle adaptatif diminue ou augmente l'activité de distribution en s'adaptant à l'état du système, par la modification éventuelle de sa politique.

Les objectifs d'ADAJ se situent dans ce domaine de la distribution dynamique de la charge, le reste de la thèse utilise le terme *distribution de charge* dans cette signification.

### 10.2.1 Les types de distribution de charge

La distribution de charge est décrite dans la littérature [Bub96, Bad00, Cav99] sous différents types, selon l'objectif à atteindre :

- *le partage de charge*, qui cherche à éviter qu'il y ait des ressources inutilisées pendant l'exécution de l'application,
- *l'équilibrage de charge*, qui prévoit une distribution équitable de la charge.

Le partage de charge est en général une distribution à *gros grain* : la charge est déplacée uniquement vers les machines qui n'ont pas du tout de charge. L'état d'une machine est ainsi considéré binaire : soit occupé soit libre. L'équilibrage de charge est une distribution de charge à *grain fin*, qui essaie d'assurer que la charge, sur toute machine, vérifie un critère d'équilibre entre toutes les charges des machines. La politique d'équilibrage de charge est supposée s'appliquer jusqu'à ce que le critère d'équilibre soit atteint.

### 10.2.2 Propriétés d'un mécanisme de distribution dynamique de charge

Les algorithmes de distribution de charge dynamique présentent plusieurs caractéristiques importantes. La liste suivante contient ces propriétés, sans être exhaustive.

- *la transparence* : la distribution de charge ne devrait pas concerner l'utilisateur. La stratégie doit rendre transparentes les opérations de distribution, aussi bien pour les autres entités, en particulier pour celles accédant à l'entité transférée, que pour l'entité elle-même.
- *l'efficacité* : l'efficacité d'un mécanisme de distribution est influencée par deux facteurs,
  - les coûts de communication engendrés par le mécanisme,

  - le coût du traitement pour extraîre la charge.
- *les heuristiques utilisées* : les heuristiques sont importantes parce qu'il n'existe pas de solution optimale dans le mécanisme d'équilibrage de charge dynamique.
- *l'adaptabilité* : l'adaptabilité offre la possibilité pour l'état courant, de s'adapter aux fluctuations apparues (ceci n'est pas le cas dans un mécanisme statique). C'est la migration qui offre cette adaptabilité aux fluctuations, parce qu'en déplaçant, sur d'autres machines, des entités en exécution, la charge est transférée.
- *la stabilité* : la stabilité est définie comme la possibilité, pour le mécanisme, de détecter ses actions qui n'améliorent plus l'état courant. L'intérêt consiste à chercher à éviter des effets de "ping-pong" (voir la section 10.3.3) liés au déplacement de la charge et, en conséquence, de supprimer des communications inutiles.
- *l'hétérogénéité* : le mécanisme de gestion des ressources doit tenir compte de l'hétérogénéité des ressources d'un réseau. Leur modélisation uniforme s'impose afin que les applications s'exécutent malgré cette hétérogénéité.

## 10.3 Gestion de la distribution de charge

La distribution de charge soulève des problèmes liés à la politique d'ordonnancement. Ces questions concernent :

- la représentation et la quantification de la charge d'une machine, souvent en termes de métriques,
- la diffusion entre les machines des informations sur la charge,
- le choix des machines susceptibles de participer à une nouvelle distribution de la charge,
- l'initiative du transfert de la charge,
- la charge qui est transférée entre les machines,
- le choix du partenaire du transfert, en rapport avec la politique de localisation,
- le moyen de transfert.

Les réponses à ses questions se retrouvent dans les différents politiques et mécanismes présentés par la suite.

### 10.3.1 Métriques de charge

La métrique de charge est importante dans un schéma de distribution de charge, car elle mesure la charge à distribuer, et devient critique pour les décisions ultérieures. Différentes métriques permettent de quantifier cette charge :

- la longueur de la queue CPU,
- l'utilisation des ressources,
- le temps de réponse des services,
- les liens de communication.

#### La longueur de la queue CPU

La longueur de la queue CPU est liée au nombre de processus en attente d'exécution (d'avoir la CPU). Une queue de longueur zéro signifie qu'aucun processus n'a attendu la CPU, et inversement, plus la queue CPU est longue, moins la CPU est disponible pour satisfaire les demandes des utilisateurs. Eagar [ELZ86] utilise la longueur de la queue CPU pour déterminer la charge d'une machine.

### L'utilisation des ressources

Une technique classique pour définir la charge est d'estimer les ressources disponibles d'une machine, et les exprimer uniformément pour pouvoir les comparer.

L'information sur la disponibilité des ressources peut provenir de différentes sources : identifier la présence d'une ressource, ou déterminer si la ressource est utilisée par une tâche, dans le système.

La plupart des systèmes considèrent comme ressource la CPU ([TLC85, Zho88]), tandis que d'autres considèrent des combinaisons de plusieurs ressources, pour prendre en compte les opérations d'entrées/sorties, et ceci contredit la règle "plus la queue CPU est longue, plus le système est chargé". Ferrari et Zhou [FZ87] comparent des métriques comme la longueur brute de la queue CPU (à un instant donné) avec des combinaisons linéaires des longueurs des queues de ressources. Leur conclusion est que la combinaison linéaire des longueurs de queues exponentiellement lissées, offre les meilleurs résultats. T. Kunz [Kun91] démontre que dans le cadre d'un automate d'apprentissage stochastique, la combinaison simple des queues CPU, mémoire et entrées/sorties, donne également de bons résultats.

Ces remarques montrent que la quantification de la disponibilité des ressources est importante et la difficulté réside dans leur calibrage.

### Le temps de réponse des services

Le temps de réponse des services quantifie l'intervalle de temps pris par un système pour traiter une demande de l'utilisateur (depuis son envoi jusqu'à la réception des résultats). Une supposition raisonnable est de classer comme libre(s), ou pas fortement chargée(s), la (les) machine(s) qui répond(ent) le plus vite à une demande [BF81, CLL85]. Dans le système V [TLC85], quand un transfert est initié, les machines sont testées à travers une requête multiple et la machine qui répond le plus vite est choisie.

Cette métrique prend en considération, non seulement, le taux de charge de la machine exécutant la demande de service, mais aussi, indirectement, le débit du réseau et l'occupation du trafic réseau. Corrélée avec l'utilisation des ressources, cette mesure peut faire la différence entre un système chargé et une communication encombrante.

### Les liens de communication

Une autre mesure peut être la charge de communication entre les entités. Pour réduire le temps de communication, il faut regrouper les entités fortement communicantes.

Les métriques de charge présentées sont généralement issues des informations d'observation d'une application. Ces informations concernent la demande de capacité de calcul et s'intéressent :

- au taux d'utilisation de CPU par demande,
- au nombre moyen de demandes, servies par une machine,
- au temps moyen de traitement d'une demande,
- à la capacité mémoire nécessaire,
- à la capacité d'une réponse (en taille des paramètres et du résultat).

Leur analyse et les décisions qui en résultent font l'objet des politiques présentées ci-dessous.

### 10.3.2 Politique d'information sur la charge

Une classification possible des politiques d'information sur la charge peut être réalisée en fonction du type d'architecture utilisée. Si l'état global d'un système distribué est connu par une seule machine, qui, alors, prend aussi la décision relative à une nouvelle distribution, cette solution est *centralisée* ; mais si plusieurs machines participent à la décision, la solution est *distribuée.*

#### Architecture de l'algorithme d'information

**Dans le modèle centralisé,** la collecte et la gestion des informations sur la charge, aussi bien que la décision sur les machines impliquées dans la distribution, ont lieu à un seul endroit. Cette méthode présente l'avantage de la simplicité et offre une vue globale de l'état de charge de chaque machine. Par contre, cette technique ne supporte pas les défaillances de la machine centrale et est tributaire des délais de transfert de l'information et des pertes des messages.

Un inconvénient plus grave concerne le comportement du modèle au changement d'ordre de grandeur de l'environnement. Ce changement, appelé aussi *l'effet d'échelle* ou *passage à l'échelle*, est reflété par l'augmentation des ressources disponibles, notamment le nombre de machines et désigne la capacité à traiter des plus gros volumes d'information en conservant une complexité du même ordre.

Pour le modèle centralisé, des goulots d'étranglement, limitant la circulation des messages vers la machine centrale, apparaissent, ce qui rend difficile le passage à l'échelle[1]. Une solution pour remédier à ce problème est de considérer des voisinages limités d'un nombre de machines, et de hiérarchiser la méthode, en l'appliquant sur les voisinages, de la même façon qu'à l'intérieur des voisinages.

**Dans le modèle distribué,** la collecte et la gestion des informations de charge concernent plusieurs machines. La décision ultérieure de distribution est prise, soit suite à une connaissance de l'état de l'environnement (par diffusion d'informations), soit suite à une décision locale, propre, non influencée de l'extérieur. La première solution risque de soulever des problèmes de trafic réseau important ou de non-utilité des messages transmis (pour les machines non-concernées dans la distribution).

Au contraire, ce modèle de transmission d'informations est plus fiable vis-à-vis des pannes systèmes. Cependant, la nature distribuée de l'information implique une gestion plus complexe.

#### Acquisition des informations

La politique d'information devrait gérer également l'acquisition des informations distantes, qui peut être réalisée de trois manières différentes :

- *politique périodique* : l'échange d'informations est réalisé périodiquement. Cette solution fonctionne en mode centralisé ou distribué. La périodicité impose un paramétrage de la fréquence d'envoi des informations. Cette fréquence peut être fixée au préalable, par des expérimentations, ou modifiée dynamiquement, en fonction des nécessités. Peu de stratégies adaptent leur période à l'évolution de la charge du système [MDLT96].
- *politique sur demande* : l'échange d'informations est réalisé à la demande de certaines machines. Cette solution est généralement utilisée en mode distribué, pour tester un changement de comportement d'une machine particulière. Elle a l'inconvénient d'induire souvent un temps d'attente entre le moment de la demande et la réception de l'information.

[1] situation caractérisée aussi comme non-scalable

- *politique de changement de l'état* : la transmission de l'information est réalisée à chaque changement significatif de l'état, à l'opposé de la politique précédente, où l'information était transmise suite à une requête. La décision sur l'importance du changement reste locale, adaptée à chaque machine.

### 10.3.3 Politique de décision

La politique de décision consiste à déterminer quelles sont les machines qui participeront à une nouvelle distribution. La sélection classe les machines en *source* et *destinataire* pour un transfert. Cette politique est basée sur la notion soit de *seuil*, soit de *transfert relatif*.

#### La politique de seuil

La politique de seuil est construite sur le principe qu'une décision de transfert de charge, qui est effectué entre deux machines, dépend de la comparaison par rapport à un seuil, des charges des machines.

Cette technique classe une machine comme *sous-chargée* ou *faiblement chargée* si sa charge est inférieure au seuil ; la machine sera *sur-chargée* ou *fortement chargée*, dans le cas contraire. La technique de *seuil unique* est très sensible à de faibles variations de charge qui engendrent des transferts successifs de charge entre les mêmes machines. C'est le cas d'un objet transféré d'une machine à l'autre, puis transféré dans le sens inverse, phénomène appelé "effet ping-pong" [Har86, RM90] ou d'un objet transféré sur une machine, et, toute de suite après, sur une autre, phénomène appelé "effet boomerang".

La solution pour contourner ces problèmes est de considérer un algorithme à *double seuil*, qui range les machines en trois classes : *sur-chargées*, *normales* et *sous-chargées*. Les machines normales ne participent pas à la nouvelle distribution. Les deux seuils ont le rôle suivant : le seuil inférieur établit l'état de la machine à partir duquel elle peut commencer à recevoir des charges et le seuil supérieur établit l'état à partir duquel la machine peut commencer à envoyer des charges.

La technique de double seuil est largement utilisée dans les algorithmes de distribution de charge [Zho88, Fol92, LK87].

Cette politique permet d'éviter l'instabilité de la politique à seuil unique. Pourtant, il existe, encore, un risque, de passer d'un état sur-chargé à un état sous-chargé ou inversement, si la quantité de charge transportée est trop importante. Ce risque est beaucoup plus faible que dans le cas précédent.

Déterminer les seuils est un enjeu majeur à cause des conséquences liées à la classification des machines. Certaines stratégies, comme celle de Eagar [ELZ86], proposent des seuils fixes, ce qui est difficilement exploitable pour tout système et application. D'autres stratégies font varier le seuil dynamiquement [Mel97], afin d'adapter l'exécution à l'évolution de la charge et donc à l'évolution globale du système.

#### La politique du transfert relatif

La politique du transfert relatif peut être définie comme un cas particulier de la politique à double seuil. Elle considère la différence de charge entre des machines et les machines dont la différence de charge dépassent un certain seuil sont prises en compte dans le transfert.

La comparaison entre les charges détermine une classification réalisée en fonction de l'état d'autres machines. Dans la politique de seuil, une machine est classée uniquement à partir de son état. L'environnement formé par les machines comparées peut être :

- l'ensemble du système,
- une partie du système.

Analyser la charge d'une machine par rapport à toutes les autres machines du système impose un surcoût important, contre-partie d'une exactitude du résultat. Ce surcoût est d'autant plus important que le nombre de machines est grand, à cause du nombre de messages échangés. Pour contourner cet inconvénient, l'analyse peut être réalisée uniquement sur un voisinage de la machine, réduisant de cette façon le nombre de messages transmis dans le réseau.

Le problème général du seuil est de le fixer (au départ, au moins), afin qu'il puisse être générique pour toute application. Eagar [ELZ86] constate que le seuil devrait être fixé à une valeur minimale au départ, pour atteindre une situation d'équilibre, et augmenté dès que l'équilibre est réalisé, pour éviter d'engendrer des activités de distribution qui n'améliorent pas l'état courant.

### 10.3.4 Mode d'initiative

Une autre classification des algorithmes de distribution de la charge est fondée sur la politique d'initiative de la (ou des) machine(s) qui décide(nt) le transfert de charge.

- *la politique d'initiative de la source* (*sender-initiative*) donne l'initiative de décision d'un transfert à la machine sur-chargée. Les machines se trouvant dans une situation de charge importante décident de participer à la distribution de la charge comme machine source.
- *la politique d'initiative du récepteur* (*receiver-initiative*) donne le choix d'accepter de la charge aux machines sous-chargées. Ces machines, se trouvant dans une situation de charge faible, décident de participer à la distribution de la charge, comme machine destinataire.
- *la politique symétrique* (*symmetrically-initiated*) est une combinaison des deux politiques précédentes, suite à la remarque faite par Eagar [ELZ85]. Il a comparé la politique d'initiative de la source avec la politique de l'initiative du destinataire et a conclu que la première est meilleure pour les systèmes aux charges faibles à moyennes, et la deuxième est meilleure pour les systèmes à charges importantes. Cette politique implique la recherche des destinataires par les sources et la recherche des sources par les destinataires.
- *la politique aléatoire* choisit de manière aléatoire une machine du système pour être la machine à initier le transfert [GT98].

### 10.3.5 Politique de sélection des entités transférables candidates

Un candidat est une entité qui sera transférée d'une machine source à une machine destinataire. Trois approches principales se distinguent, pour leur sélection :

- choix d'un candidat quelconque,
- choix d'un candidat raisonnable,
- choix d'un bon candidat.

#### Choix d'un candidat quelconque

Dans cette approche, il n'existe pas de classification des entités transférables, toutes étant considérées éligibles. Le choix se fait de manière aléatoire, ce qui apporte de la simplicité et un surcoût minimum à l'algorithme.

#### Choix d'un candidat raisonnable

La méthode de choix aléatoire du candidat peut paraître efficace, mais elle peut aussi avoir des résultats inattendus. Par exemple, le temps de transfert d'une entité doit généralement être

compensé par un gain de temps sur l'attente et l'exécution des traitements sur cette entité. Si ce n'est pas le cas, le transfert ne s'avère pas intéressant. Ainsi, tous les candidats ne seront pas considérés, mais seulement ceux qui présentent de l'intérêt. L'écartement des candidats non intéressants peut être fait soit :

- statiquement, par l'analyse, à la post-exécution, du comportement de l'application,
- dynamiquement, par l'enregistrement du comportement à l'exécution.

### Choix d'un bon candidat

Si l'approche précédente écarte les candidats qui ne présentent pas d'intérêt, d'autres critères peuvent être imposés pour choisir un bon candidat.

Certains systèmes choisissent les entités dont la durée de vie est la plus longue, estimée au moment de la sélection. Par exemple, pour les processus, ce choix est fait en supposant que plus un processus a consommé de temps CPU, plus il a la chance de continuer à s'exécuter pendant une période de temps importante. C'est le choix fait par Chorus [OTCH94].

D'autres systèmes proposent une stratégie inverse, autorisant le transfert des processus nouvellement créés. Ce critère part de l'hypothèse que les processus nouvellement créés se terminent plus tard que ceux créés auparavant. Cette technique est utilisée dans le système V [TLC85], où seuls les processus jeunes (créés récemment) sont candidats au transfert.

Plusieurs critères peuvent être considérés. Leur évaluation simultanée utilise la technique d'agrégation, décrite brièvement ci-dessous.

### L'agrégation des critères

Plusieurs techniques d'agrégation existent, parmi lesquelles, les trois plus fréquentes [Phi00] sont résumées ici :

- *la procédure lexicographique* définit un ordre d'importance au sein des critères, permettant de les ordonner du plus important au moins important. L'évaluation d'un premier critère ordonné donne le choix du candidat, et en cas d'égalité le deuxième critère sera considéré. Le principe est itéré jusqu'à la fin des critères. Il s'agit d'un mode de choix hiérarchique qui est simple à mettre en œuvre et n'introduit pas un surcoût de calcul important. Il se base sur la classification des critères, sans tenir compte des compensations éventuelles d'un critère par un autre.
- *la procédure sommative pondérée* associe, à chaque critère, un coefficient d'importance et réalise la somme pondérée des critères par les coefficients. Cette technique est basée sur le principe de compensation des critères. Par contre, elle impose la nécessité de pouvoir exprimer l'ensemble des critères dans une plage de valeurs et de grandeurs homogènes. La détermination des coefficients d'importance est également un problème majeur, qui influence le résultat de l'agrégation.
  Il est nécessaire que la contribution de chacun des critères à l'agrégation soit indépendante des valeurs des autres critères, pour pouvoir utiliser la procédure sommative [Fis78].
- *la procédure multiplicative pondérée*, similaire à la procédure sommative, consiste à réaliser le produit pondéré des critères. Cette technique est plus adaptée aux critères suivant une distribution exponentielle.

### 10.3.6 Politique de localisation

L'objectif de la localisation est de trouver un partenaire pour le transfert, c'est-à-dire, une autre machine prête à recevoir ou à donner de la charge supplémentaire. Evidemment, cette machine sera une machine sous-chargée, pour une politique à l'initiative de la source ou une machine sur-chargée pour la politique à l'initiative du récepteur. Le choix entre plusieurs machines de ce type peut être aléatoire ou basé sur des critères.

### 10.3.7 Le moyen de transfert

La modalité de transfert de charge quand une nouvelle distribution de charge est nécessaire, détectée et demandée par les politiques précédentes, est la *migration.* Cette opération consiste à déplacer une entité depuis une machine source, vers une machine destinataire, afin de transférer la charge attachée à cette entité.

#### Granularité de la migration

L'entité à transférer est directement liée à la granularité de traitement, qui est l'unité de travail dans un système. En fonction des systèmes et des langages de programmation, les granularités existantes, variant de celles à grain fin à celles à gros grain, sont :

- les instructions,
- les appels distants (RPC[2]),
- les threads,
- les objets,
- les processus.

#### Alternatives à la migration

Les alternatives à la migration sont le *placement initial* et l'*exécution distante.*

**Le placement initial** consiste à affecter à une entité, lors de sa création, une machine d'accueil. Le placement peut être employé en conjonction avec la migration, pour éviter d'avoir au départ ou pour corriger une mauvaise distribution.

**L'exécution distante** consiste à invoquer, sur une autre machine, donc utilisant des ressources non-locales, une partie de code. La méthode a l'avantage d'être moins coûteuse que la migration, grâce à sa simplicité. Cette technique rappelle le placement des requêtes, qui a le but d'assigner des requêtes aux serveurs, pour améliorer la performance du système. L'assignation des requêtes est un aspect important dans les environnements à objets répliqués, où les requêtes des clients peuvent être servies par plusieurs objets.

#### Migration versus placement initial

La migration considère qu'une entité peut être *active*, pour laquelle une activité s'exécute, ou *passive*, pour laquelle il n'existe pas d'activité courante. Ces deux types d'entités révèlent deux types de migration, la *migration restreinte*, et la *migration d'une activité* :

[2]Remote Procedure Call

- la migration restreinte envisage, pour la migration, des entités qui n'ont pas d'activité commencée. C'est le cas adapté pour les systèmes non-préemptifs, qui n'interrompent pas l'activité en exécution.
- la migration d'une activité nécessite un mécanisme de préemption des activités en exécution, avec le risque d'augmenter le surcoût de la mise en œuvre.

Dans les systèmes préemptifs ou non-préemptifs, vues les contraintes de migrations imposées, une autre possibilité pour la distribution de la charge est envisagée, *le placement.* Le placement est une alternative à la migration, pour la distribution de charge, qui évite un surcoût important de mise en œuvre de la migration et conserve le principe de non-interruption des activités pour les systèmes non-préemptifs.

Le placement consiste à trouver une machine destinataire pour une entité avant qu'elle soit créée, donc, avant de démarrer son activité. Un des facteurs importants qui joue un rôle dans la décision entre un placement initial et une migration est le coût. Le placement est moins coûteux que la migration parce que lors de la migration, la récupération et le transfert de l'état de l'exécution est complexe et introduit un surcoût non-négligeable. Le placement ne génère pas le même coût, parce que l'activité de l'entité n'est pas encore démarrée.

## 10.4 Synthèse

Dans ce chapitre, nous avons présenté la problématique de distribution de charge dans les systèmes distribués. Les objectifs les plus couramment visés de la distribution de charge sont la minimisation du temps de réponse des applications, la minimisation du temps d'inoccupation des processeurs ou l'augmentation de la fiabilité du système.

Deux grands axes sont considérés : le modèle de distribution et la gestion de la distribution. Le modèle permet de classer les algorithmes en une des trois classes : statique, dynamique ou adaptative, avec l'objectif visé, du partage ou d'équilibrage de charge. La gestion de la distribution concerne essentiellement les modèles dynamique et adaptatif, proposant une classification des politiques qui composent un système de distribution de charge.

La même problématique est soulevée dans le cas particulier des systèmes distribués à base d'objets. Cette fois, l'unité de distribution est l'objet. Dans ces systèmes, la distribution de charge peut être toujours réalisée au niveau processus, car les objets sont encapsulés dans des processus pour pouvoir s'exécuter. Pourtant, des caractéristiques supplémentaires du modèle d'objets, comme la granularité de l'objet ou le degré d'activité de l'objet doivent être prises en compte.

Dans le chapitre suivant, nous présentons des exemples de systèmes distribués à base d'objets qui offrent des facilités pour réaliser la distribution de charge.

# Chapitre 11

# Implémentations des algorithmes de distribution de charge dans les systèmes à base d'objets

L'intégration des mécanismes de distribution de charge dans les systèmes distribués objets soulève des questions supplémentaires liées à l'entité de distribution, la granularité de distribution ou le type de distribution réalisée.

Dans les systèmes orientés objets, la distribution de charge peut toujours être réalisée au niveau processus, car les méthodes appelées sur les objets s'exécutent dans des processus. Une autre forme de distribution est réalisée au niveau objet. L'unité de distribution est, dans ce cas, soit l'objet, soit l'invocation d'une méthode d'un objet. Les approches de distribution dans les systèmes à base d'objets sont : le placement d'objets, la migration d'objets et l'assignation d'invocations de méthodes.

La granularité de la distribution d'objets n'est pas moins importante que celle d'un processus, cela dépendant de la quantité de calcul réalisé par un objet. Pour acquérir une granularité plus grossière, certains systèmes [Jen96, CHPB96] utilisent le concept de *grappe* (*cluster*), pour grouper des objets. La grappe devient l'unité de distribution dans ces systèmes.

Le type de distribution des objets par migration dépend de l'état de l'objet lors de la migration : si un objet migre pendant qu'il y a des méthodes en cours d'exécution, la migration est *préemptive*, dans le cas contraire, la migration est appelée *non-préemptive*. Beaucoup de systèmes font le choix d'une migration non-préemptive, vu le coût important et la complexité de la migration préemptive. De plus, certains systèmes ne considèrent que la distribution de nouvelles entités créées, d'où l'utilisation unique d'une politique de placement initial.

Dans ce chapitre, des exemples de systèmes distribués objets qui offrent des mécanismes de distribution de charge sont présentés. Leurs modèles de programmation et d'exécution sont brièvement décrits, et des détails concernant le mécanisme de distribution sont donnés.

## 11.1 Guide-2

Guide-2[1] [Jen96] est un prototype de système d'exploitation, conçu pour le travail coopératif et le partage d'objets persistants dans un environnement de stations de travail, appelées *nœuds*, connectées en réseau.

[1]Guide = Grenoble Universities Integrated Distributed Environment, développé à Bull-IMAG

**Les modèles de programmation et d'exécution.** Les objets en Guide-2 sont vus comme des serveurs dans une architecture client-serveur, tout objet ayant un espace local d'adresses et des interfaces pour la communication. Les objets sont groupés dans des *grappes d'objets*, qui constituent l'unité de distribution et de placement. Le placement des objets tient compte de la localité de référence, c'est-à-dire que les objets placés dans la même grappe interagissent, par des invocations locales de méthodes.

Le modèle d'exécution est basé sur la notion de *tâche*, comme espace distribué d'adresses virtuelles, partagé par des threads de contrôle appelés *activités*. Les applications Guide-2 sont généralement implémentées comme une tâche ayant une ou plusieurs activités.

**La distribution de charge** s'appuie sur le placement des objets, vue l'absence de la préemption des activités. L'indicateur de charge utilisé est la longueur moyenne de la queue des tâches pendant les 5 dernières secondes. Cette mesure quantifie uniquement la charge des processeurs et pas celle de l'application. La politique d'information est centralisée, par la collecte d'informations de charge et la redistribution d'un vecteur de charge à tous les nœuds. La politique de décision utilisée est à double seuil, le mode d'initiative étant à la source (le nœud sur-chargé) ou la destination (le nœud sous-chargé). La politique de sélection considère uniquement les grappes nouvellement créées, correspondant à un placement de grappes. Guide-2 envisage une amélioration, en éliminant les grappes qui n'apporteront pas d'amélioration dans la nouvelle distribution. Cette technique rappelle le "filtre historique" de Anders Svensson [Sve90], qui enregistre le temps moyen de l'application et les applications, dont le temps d'exécution est le plus long, sont sélectionnées. Un nœud entre les moins chargés est choisi comme destination dans la politique de localisation, et la technique de tourniquet est employée si plusieurs nœuds avec les mêmes caractéristiques existent.

## 11.2 COOL v2

COOL v2 [CHPB96] est développé au Laboratoire d'Informatique de Besançon, à l'Université de Franche-Comté. Réalisé au-dessus du système COOL, par extension du langage C++, COOL v2 introduit des notions de distribution, persistance et interopérabilité. Le système COOL (Chorus Object Oriented Layer) est une infrastructure pour développer des applications réparties à base d'objets, au-dessus du micro-noyau Chorus.

**Les modèles de programmation et d'exécution.** L'espace d'adressage est structuré en grappes, qui regroupent des objets. Un objet alloué à une grappe ne peut pas changer de grappe dynamiquement. La grappe constitue la granularité de migration, directement corrélée avec le coût de l'opération exécutée sur la grappe.

**La distribution de charge.** L'indicateur de charge est défini à partir d'informations statiques, comme la puissance du processeur, ou la capacité mémoire, et d'informations dynamiques, comme le nombre d'activités en cours d'exécution ou les relations entre les objets. Le taux d'utilisation du processeur est calculé à partir du nombre d'activités en cours, ce qui évalue la charge du site. En même temps, les relations entre les objets, en termes de communication, sont quantifiées dans un modèle relationnel.

La distribution de charge se décline sous deux formes, en fonction du type de déséquilibre détecté. Si une homogénéité des charges est souhaitée, un traitement périodique est appliqué, qui compare un indicateur de perte d'homogénéité des charges avec un seuil donné. Si un déséquilibre

important est détecté, un traitement exceptionnel est invoqué qui détecte des situations extrêmes, de sous-charge ou/et de sur-charge. Dans ces cas, la politique de décision est à double seuil, avec un mode d'initiative symétrique. La politique de sélection choisit la grappe qui modifie le moins possible la structure relationnelle et engendre des traitements. La politique de localisation choisit un site destinataire de manière aléatoire.

## 11.3 Amadeus

Amadeus [TC92], développé au département d'Informatique de l'Université de Dublin, est un environnement pour une programmation distribuée persistante qui vise à imposer de changements minimaux à des langages objets existants (C++, Eiffel) afin d'offrir un support simple pour la programmation distribuée.

**Les modèles de programmation et d'exécution.** Un des modèles de programmation propose une extension du langage C++ (C**) pour marquer des classes persistantes et globales. Les objets sont regroupés en grappes, comme dans COOL-v2, qui sont placées une seule fois sur un nœud. Le principe de traitement considère des tâches, composées d'un ensemble d'activités. Une activité coïncide à un thread de contrôle distribué.

**La distribution de charge** consiste à attacher un nœud préféré à la création d'une activité, ou à placer des objets, pour les grappes déjà allouées. L'indicateur de charge utilisé est le nombre d'activités sur un nœud, qui est diffusé de manière centralisée. La politique de décision choisit un nœud aléatoirement, et sa charge est augmentée d'un facteur, afin d'éviter d'être submergé par des décisions ultérieures, avant que l'effet du placement actuel soit pris en compte.

## 11.4 Isatis

Isatis [BBI+94], développé à Rennes, IRISA, propose un mécanisme pour améliorer les performances d'applications orientées objets, distribuées, en implémentant le placement des exécutions de méthodes, qui s'adapte à la charge du processeur et aux caractéristiques des objets (leur taille ou la taille des paramètres des méthodes).

**Les modèles de programmation et d'exécution.** Isatis est développé pour les applications C++ s'exécutant sur l'environnement Unix et Mach.

**La distribution de charge** suit une politique adaptative, appliquée uniquement au placement initial des exécutions de méthodes. En Isatis, les exécutions ne migrent pas. L'indicateur de charge utilisé est le temps pris par l'exécution d'une méthode, comprenant à la fois, le temps de transfert de l'appel, le temps d'attente dans la queue du processeur avant que la méthode soit lancée, le temps effectif d'exécution et le temps pour le retour vers l'appelant. La prise en compte de la charge locale de la machine, à part la charge générée par l'application, est réalisée par le calcul du temps d'attente dans la queue des tâches prêtes à s'exécuter dans la queue Unix. La considération de toutes ces actions, en temps d'exécution, utilise la théorie multi-critère de Fishburn [Fis78], pour minimiser le cumul des temps de chaque opération. La politique d'information des charges locales est périodique, par diffusion de messages. La politique de sélection classe les machines en fonction d'un critère d'éligibilité et utilise la technique de seuil. Lors du placement, la sélection réalisée est

aléatoire. Cependant, une autre technique est mentionnée, qui choisit comme machine destinataire celle qui a le plus grand nombre d'objets référencés par la méthode, dans l'intention de minimiser les communications distantes.

## 11.5 ABACUS

ABACUS [APGG00], développé à l'Ecole d'Informatique de l'Université Carnegie Mellon, propose un placement optimal pour les applications C++ dans une grappe de clients et serveurs, dans le contexte de variations dynamiques dans le comportement de l'application, dans la disponibilité des ressources et dans la charge.

**Les modèles de programmation et d'exécution.** Le modèle de programmation utilise des *objets mobiles* comme des entités de granularité moyenne, sources de calcul intensif. Ces objets ont un identifiant unique dans tout l'environnement, acquis lors de leur création. Les objets mobiles peuvent migrer, de manière non-préemptive, dans le sens où les nouveaux appels venant vers l'objet à migrer seront bloqués, et les appels courants seront terminés, avant de démarrer la migration. Dès que la migration est finie, les appels bloqués sont délivrés. La gestion des communications lors de la migration n'est pas assurée par le mécanisme de redirection de messages, mais par la redirection des messages vers le site d'origine de l'objet.

D'autres objets, appelé *objets de stockage*, sont utilisés, qui ne sont pas migrables, mais attachés au site initial de création. A l'exécution, l'application est représentée comme un graphe d'objets mobiles communicants, où les sommets sont les objets mobiles et les arcs les invocations de méthodes.

L'environnement d'exécution observe et change dynamiquement le placement des fonctions pour les applications qui manipulent de larges ensembles de données. L'observation de l'utilisation des ressources et de communication entre les objets donne des informations sur les patterns de communications les plus importants et sur les nécessités de ressources par objet. Les composants de l'environnement d'exécution sont :

- un *manager de localisation* (*binding manager*), composant de migration et d'invocation transparente, qui est responsable de la création des références transparentes sur les objets mobiles, et de rediriger les invocations ou déclencher la migration,
- un *manager de ressources* (*resource manager*), composant d'observation et de gestion des ressources, qui collecte des statistiques sur la charge locale et sur le temps d'attente dans le transfert réseau.

L'observation de l'utilisation des ressources regroupe des informations sur :

- le nombre d'octets transférés entre les objets, par l'analyse des paramètres,
- la consommation mémoire (le taux de mémoire allouée dynamiquement par objet),
- le nombre d'instructions exécutées pendant une période de temps,
- le temps d'attente d'une méthode pour s'exécuter, par la modification d'un appel système Unix.

**La distribution de charge.** L'indicateur de charge tient compte des informations du manager de ressources spécifiées ci-dessus, corrélées avec la charge courante et la vitesse du processeur. La politique de décision se base sur des critères de minimum ou maximum, en supposant que pendant la période de temps courante, l'histoire de la période de temps précédente se répète. La sélection des objets mobiles à migrer est basée sur la technique de seuil. La différence entre l'avantage de

déplacer et le coût de déplacement mesure le gain net de l'opération, le seuil ayant pour but d'éviter les migrations qui apportent de petites améliorations.

## 11.6 LOCA

LOCA [Rac97], développé à l'Institut d'Informatique, de l'Université Technique de München, intègre et évalue les performances de différentes stratégies de distribution de charge dans les environnements CORBA.

**Les modèles de programmation et d'exécution.** Le modèle de programmation proposé est du type exportateur/importateur/courtier (*Exporter/Importer/Trader model*) qui étend le modèle client-serveur. Le courtier joue le rôle d'intermédiaire des services entre les clients et les serveurs. La médiation des services contient les étapes suivantes :

- les serveurs qui veulent offrir des services à ses clients exportent ces services, en enregistrant leur fonctionnalités auprès du courtier,
- si un client veut utiliser un service particulier, il appelle une opération d'importation du courtier, qui lui renvoie un serveur adéquat ou un service demandé,
- pour utiliser le service, le client appelle directement le serveur récupéré et lui passe la demande.

Les composants de l'architecture LOCA sont :

- les serveurs et les services,
- le courtier,
- les clients,
- le superviseur, qui est responsable de la collection des informations de charge, décide si une migration est nécessaire et contrôle la migration.

**La distribution de charge.** Les niveaux d'abstraction auxquels la distribution de charge peut être réalisée sont :

- *objet*, où la distribution peut être effectuée soit au courtier, qui sélectionne une référence d'objet et la retourne vers le client, soit au client, si le courtier retourne toutes les références d'objets adéquats et le client décide laquelle utiliser.
- *implémentation*, où la distribution de charge consiste à associer des instances d'objets qui offrent des services à un serveur adéquat (les instances d'objets peuvent migrer entre les serveurs).
- *système*, où la distribution signifie associer les processus aux nœuds et les requêtes aux ressources.

La granularité de la distribution de charge peut être soit fine, au niveau des requêtes, soit moyenne à grosse, pour les objets services. L'indicateur de charge est l'utilisation du serveur, c'est-à-dire le pourcentage de temps où le serveur est occupé pendant les $k$ dernières secondes. La valeur $k$ détermine comment les changements rapides sont reflétés dans l'indice de charge. La politique d'information est centralisée, périodique, avec plusieurs modes d'initiative :

- source, réalisée de manière décentralisée par les clients, par l'association de nouvelles requêtes à un serveur approprié,
- récepteur, réalisée à l'initiative des serveurs, par la distribution de nouvelles requêtes,
- mixte.

La politique de localisation est soit aléatoire, à base de seuil, le serveur choisi étant celui avec la plus petite charge entre un nombre de serveurs choisis aléatoirement, soit statistique, la probabilité étant proportionnelle au poids du serveur (politique centralisée au niveau du serveur).

## 11.7 LoDACE et LLS

LoDACE (Load Distribution Architecture for Distributed Object Computing Environments) [Bad00], développé à l'Université de Montreal, Canada, est une architecture pour la mise en œuvre des stratégies d'assignations de requêtes pour effectuer la distribution de charge en CORBA. L'assignation de requêtes, problème similaire à celui de l'assignation d'une tâche à une collection de processeurs, est une alternative aux solutions de placement et migration d'objets, qui s'avèrent souvent, d'après les auteurs, difficiles à réaliser.

**Les modèles de programmation et d'exécution.** Les objets au niveau application sont de deux types : objets clients et objets serveurs. Les objets serveurs offrent divers types de services que les objets clients utilisent. Dans le cadre de l'application, les serveurs sont organisés dans des grappes logiques. Les serveurs d'une même grappe offrent le même type de service.

L'architecture LoDACE comporte les entités suivantes :

- des clients et des serveurs,
- une couche middleware, pour acheminer les requêtes et les données,
- un ensemble de services pour réaliser l'assignation de requêtes des objets clients aux objets serveurs.

**La distribution de charge** que LoDACE propose est réalisée au niveau opération. Elle consiste à assigner des requêtes (méthodes, invocations) aux objets appropriés qui implémentent les méthodes appelées. Trois approches possibles existent pour l'assignation de requêtes :

- **l'approche client,** basée sur le choix volontaire des clients. Un objet client peut utiliser, pour le choix du serveur cible, différentes stratégies : statiques (aléatoire ou cyclique) ou dynamiques (choix du serveur le moins chargé).
- **l'approche répartiteur,** basée sur l'utilisation d'un coordonnateur central (le répartiteur). Un composant central joue le rôle d'interface entre les clients et les serveurs, tel que les clients ne sont pas obligés de connaître les serveurs pour pouvoir recevoir les services désirés. Les requêtes des clients sont acheminées à des serveurs appropriés par le répartiteur qui implémente certaines stratégies d'ordonnancement et de distribution de charge et qui collecte l'information sur l'état des serveurs.

  La politique d'information est centralisée, périodique ou sur demande, les informations de charge étant récupérées dans un vecteur de charge. La politique de décision utilise la technique à double seuil, classant les serveurs en : sur-chargés, moyennement chargés et légèrement chargés. La politique de localisation choisit un serveur cible par l'analyse du vecteur de charge.
- **l'approche serveur,** basée sur la coopération des serveurs. Cette approche suppose que les serveurs d'une même grappe se connaissent mutuellement et coopèrent pour mettre en œuvre la distribution de charge par échange d'informations de charge. Le mode d'initiation est soit de la source, soit du récepteur. La politique de décision est basée sur la comparaison du taux d'utilisation du serveur avec un seuil donné. La politique de localisation est aléatoire, à base de seuil, en fonction du mode d'initiative : minimal (maximal), le moins chargé (le plus chargé). Seules les nouvelles requêtes sont considérées.

Le LLS (Load Sharing Service) est un service générique capable de supporter divers types de serveurs. Différentes stratégies d'assignation de requêtes sont proposées : le serveur le moins récemment utilisé (LRU[2]), le LRU relatif qui tient compte du taux de service de chacun des serveurs, et le

[2]Least Recently Used

serveur le moins chargé, pour prendre en compte la charge créée par chaque type de requête (la charge est exprimée en nombre d'instructions exécutées).

## 11.8 Charm++

Charm++ [BK99], développé au Laboratoire de Programmation Parallèle de l'Université d'Illinois, en Urbana-Champaign, est un langage orienté objets, construit au-dessus du système Converse, basé sur le passage de messages, qui supporte l'encapsulation et l'héritage dans les objets parallèles. L'intérêt du Charm++ est d'ordonnancer le travail pour pouvoir utiliser la puissance de calcul des stations partiellement disponibles.

**Les modèles de programmation et d'exécution.** Les objets en Charm++ ont une granularité moyenne ; la granularité peut être plus importante par la construction des groupes comme ensembles d'objets. Une autre construction, *le tableau* (*array*), est une collection multi-dimensionnelle de données. Les messages vers un tableau peuvent être diffusés à tous les éléments du tableau, ou vers un sous-ensemble d'éléments. Le tableau est l'entité de migration en Charm++.

**La distribution de charge.** L'indicateur de charge mesure la disponibilité du processeur, calculée par le temps CPU obtenu par un processus pendant une période de temps, divisé par le temps écoulé. En connaissant le temps d'inactivité, le temps consommé par d'autres processus peut être calculé, ainsi Charm++ connaît si le déséquilibre est dû à une charge à l'extérieur de l'application.

La politique d'information de charge est décentralisée et régulière. Lors de la phase de sélection, un ensemble d'objets qui se rapproche le plus de la fraction de la charge totale est choisi.

## 11.9 Dome

Dome [ABL+95], développé à l'Université Carnegie Mellon, Pittsburgh, est un environnement de programmation et d'exécution pour offrir aux programmeurs une interface simple et intuitive pour la programmation parallèle, en utilisant une bibliothèque de classes C++ génériques, les *classes patron* (*templates*), pour la description des entités du programme et PVM pour le contrôle de processus et la communication.

**Les modèles de programmation et d'exécution.** Les applications Dome sont modélisées sous la forme de programmes SPMD. Une classe Dome représente une collection importante d'éléments similaires, comme un vecteur d'objets. Lors de la création d'un objet Dome, les éléments sont partitionnés et distribués entre les processus du programme distribué. Les partitionnements offerts en Dome sont des directives qui indiquent si les éléments sont dupliqués sur tous les processus, divisés équitablement entre les processus, ou divisés équitablement au départ, mais avec une redistribution périodique, en fonction des décisions du mécanisme d'équilibrage de charge.

**La distribution de charge.** La distribution de charge consiste à redistribuer les objets, sous une politique d'équilibrage, pour répondre aux changements dans l'environnement d'exécution. Ces décisions sont prises à partir de l'observation de la vitesse du processeur et de la performance de communications.

L'indicateur de charge utilisé est le temps pris par chaque processeur pour le calcul (dans la dernière phase de calcul du modèle SPMD). Ce temps quantifie le taux auquel les processeurs ont exécuté le programme Dome.

La politique d'information de charge implémentée est centralisée ou distribuée. La décision de localisation pour des objets migrés est réalisée soit au niveau central, dans la politique d'information centralisée, soit au niveau de chaque processeur, pour la politique d'information distribuée. Vue la topologie virtuelle des machines, en anneau ou linéaire, le transfert des objets est réalisé uniquement entre les voisins.

## 11.10 Bilan des travaux

Le tableau suivant résume les différentes approches choisies dans les systèmes distribués à base d'objets, présentés dans ce chapitre, pour rendre la fonctionnalité de distribution de charge.

| Système | langage | partage vs équilib. | distr inter/intra appli | centr. vs distr. | IS vs IR | PO/ MO/ AR | gran |
|---|---|---|---|---|---|---|---|
| Guide-2 | syst expl | équilib. | inter | mixte | mixte | PO | grappe |
| COOL-v2 | C++ (Chorus) | équilib. | mixte | - | IS/IR | MO | grappe |
| Amadeus | C** | équilib. | inter | distr. | remapp | PO | grappe |
| Isatis | C++ (ARCHE) | équilib. | mixte | distr. | - | AR | obj |
| ABACUS | C++ | équilib. | mixte | centr. | - | MO(np) | obj mobile |
| LOCA | CORBA | partage | inter | centr. | IS/IR/ mixte | MO(np) et AR | requête/ obj |
| LoDACE | CORBA | - | inter | mixte | RI | AR | requête |
| Charm++ | C++ | équilib. | mixte | distr. | remapp | MO(np) | tableau |
| Dome | C++ | équilib. | mixte | centr./distr. | remapp | MO(np) | obj |

Tableau 11.1 – Bilan des approches de distribution de charge dans différents systèmes

Légende :
- MO = migration d'objets
- p = préemptive
- np = non-préemptive
- PO = placement d'objets
- AR = assignation de requêtes
- centr. = centralisée
- distr. = distribuée
- équilib. = équilibrage
- IS = initiative de la source
- IR = initiative du récepteur
- gran = granularité
- - = non spécifié

# Chapitre 12

# Outils disponibles en ADAJ

L'exécution efficace d'une application implique une connaissance de son comportement et de l'environnement dans lequel elle s'exécute. Le comportement de l'application concerne l'évolution des objets par rapport aux communications engendrées et aux traitements effectués. La connaissance de son comportement apporte une information utile pour le mécanisme d'équilibrage de charge *intra-application* : seulement à l'intérieur de l'application, la charge sera équitablement distribuée. Dans ce cas, un déséquilibre de charge venant de l'extérieur de l'application risque d'imposer des décisions contradictoires avec celles pour l'application.

L'environnement dans lequel une application s'exécute accueille généralement plusieurs applications. Le comportement d'une application peut exiger des transformations nuisant au comportement d'une autre application. De plus, le déséquilibre de charge peut avoir une cause externe à l'application, ayant des conséquences sur l'évolution de l'application. Le mécanisme d'équilibrage *inter-applications* est censé à corriger ce déséquilibre. Cependant, un équilibre externe de charge n'implique pas un équilibre interne pour l'application.

L'objectif est de fournir un mécanisme d'observation à la fois du comportement de l'application et de l'environnement dans lequel elle s'exécute, pour offrir une distribution équitable de la charge pour l'application utilisateur, dans le contexte d'une exécution dans un système multi-utilisateurs. Ce mécanisme d'observation doit être le plus générique possible, indépendant de la plate-forme d'exécution. Le langage Java assure ces caractéristiques pour ses applications, donc son utilisation pour la description de l'outil d'observation est judicieuse. En même temps, l'observation devrait se faire à un coût minimal, sans perturber essentiellement l'exécution de l'application observée ou des autres applications s'exécutant dans l'environnement.

Ainsi, le mécanisme d'observation en ADAJ, comporte :

- une observation de la charge de la plate-forme d'exécution,
- une observation de l'évolution de l'application.

Les deux types d'observation devraient être pris en compte pour fournir un mécanisme d'équilibrage de charge à la fois inter et intra-application.

## 12.1 Observation de l'évolution de la charge

Dans un environnement distribué, l'exécution efficace d'une application implique un équilibre entre les quantités de traitements effectués par chaque processeur. La détection du déséquilibre peut être faite si une estimation de l'occupation des machines de l'environnement existe. Cette estimation est appelée *charge*, dont la définition sera donnée dans ce chapitre. La charge est liée à la consommation des ressources (CPU, mémoire). Le mécanisme prévu en ADAJ, écrit 100 % en

Java, donne une estimation de la charge d'une machine, en vue de l'utilisation pour la détection d'un déséquilibre.

### 12.1.1 Définition de la charge

L'exécution d'une application Java implique une consommation des ressources. Les ressources qui sont utilisées consistent essentiellement en l'unité centrale et la mémoire. L'unité centrale permet l'exécution de l'application à une vitesse plus ou moins élevée, tandis que la mémoire permet la création des entités liées à l'application.

La charge mesure l'utilisation des ressources sur trois niveaux :
- de l'application, qui définit *la charge de la machine virtuelle Java* (JVM),
- de la JVM, qui définit *le potentiel de la JVM*,
- de la machine physique qui accueille la JVM, qui définit *la charge de la machine physique.*

#### Charge de la JVM

La charge de la JVM est définie comme l'utilisation des ressources par une application dans la JVM. Cette charge consiste à estimer quel est le pourcentage d'occupation de la JVM par l'application, c'est-à-dire, quel pourcentage de la puissance de la JVM est affecté à l'application, en termes de CPU et de mémoire.

Dans une application Java, la consommation de la mémoire résulte de la création d'objets.

#### Potentiel de la JVM

Le potentiel de la JVM (on dit aussi l'importance de la JVM, en tant que processus dans la machine physique d'accueil) est défini comme le pourcentage de l'utilisation, par la JVM, des ressources de la machine physique d'accueil. Cette utilisation s'exprime aussi en termes de CPU et de mémoire. Le potentiel de la JVM n'est pas lié à l'application elle-même, mais à l'environnement dans lequel le processus correspondant à la JVM s'exécute. Le système d'exploitation lui affecte, comme pour tout autre processus, le processeur et une partie de la mémoire, à l'aide du mécanisme d'*ordonnancement des processus.* Un aspect complexe lié à l'ordonnancement est la gestion des threads d'une application, qui est réalisée en fonction du système d'exploitation (voir la section 12.1.2).

#### Charge de la machine physique

La charge de la machine est définie comme le pourcentage de l'utilisation des ressources de la machine (CPU, mémoire), par tous les processus s'exécutant dans la machine physique. Cette mesure dépend donc du nombre de processus s'exécutant et de leur demande de ressources de la machine.

Les trois types de charge sont résumés dans la figure 12.1.

Les trois mesures de charge s'imposent pour les raisons suivantes (en ne considérant que l'utilisation de la CPU) :
- l'équilibrage de charge d'une application doit dépendre du potentiel de la JVM. Des charges égales pour des JVM, correspondant à des potentiels différents, ne prouvent pas un équilibre, si les JVM correspondantes sont d'importance différente dans la machine.

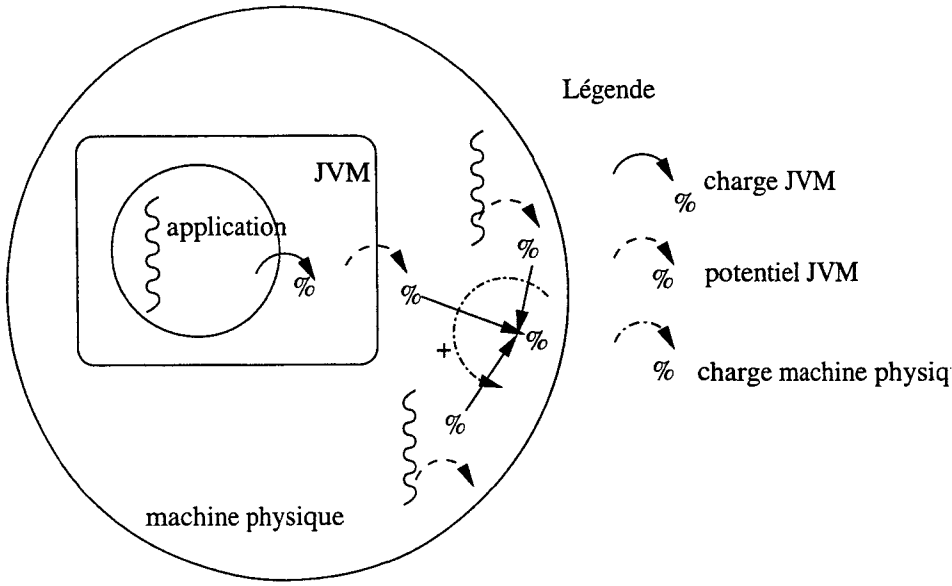



Figure 12.1 – Charges

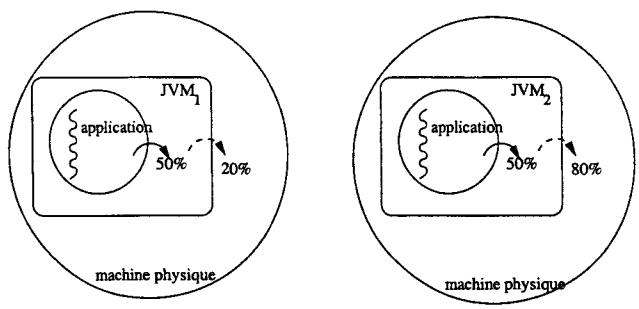



Figure 12.2 – Exemple 1 de charges

Dans l'exemple présenté dans la figure 12.2, la partie de l'application qui tourne dans la $JVM_1$, s'exécute plus lentement que celle qui tourne dans la $JVM_2$, car elle reçoit moins de temps CPU que l'autre partie de l'application.

- l'équilibrage de charge d'une application doit prendre en compte la charge de la machine physique dans laquelle la JVM s'exécute.
  Dans l'exemple de la figure 12.3, si la machine physique accueillant la $JVM_1$ est chargée à 20 % et l'autre machine physique accueillant la $JVM_2$ est chargée à 80 %, la partie de l'application qui tourne dans la $JVM_1$ a plus de chances à s'exécuter plus vite que la partie de l'application s'exécutant dans la $JVM_2$.

De plus, les caractéristiques d'une machine physique (type CPU, fréquence CPU, capacité mémoire) doivent être prises en compte, car la vitesse d'exécution et la mémoire peuvent avoir des valeurs différentes.

### 12.1.2 Gestion des threads en Java

Le problème de déterminer quel thread d'un programme est désigné pour s'exécuter à un moment donné, entre plusieurs threads demandant l'utilisation du processeur, est à la charge de

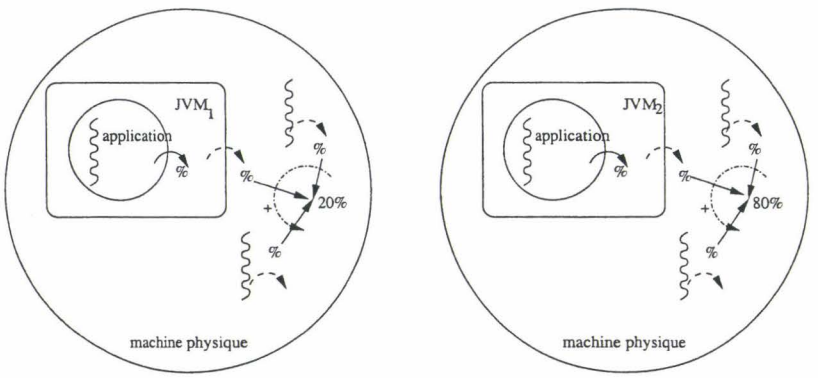



Figure 12.3 – Exemple 2 de charges

l'*ordonnanceur*. La technologie Java implémente un ordonnanceur de type préemptif, basé sur la priorité.

### Principe d'ordonnancement

Chaque thread Java reçoit une priorité (de 1 à 10), modifiable uniquement par le programmeur et non-affectée par la JVM. Le principe d'ordonnancement mis en œuvre par la JVM, est que les threads de plus haute priorité ont, en général, la préférence sur des threads de priorité inférieure [Suni]. Toutefois, une telle préférence n'est pas une garantie que le thread de la plus haute priorité sera toujours actif (appelé *thread actif courant*, ou *thread prêt*, choisi pour s'exécuter). L'ordonnanceur peut choisir un thread de priorité inférieure pour l'exécution. Ceci, afin d'éviter la "famine", situation dans laquelle un processus est retardé indéfiniment par l'action des autres processus.

La préemption est exercée par un thread qui devient prêt, ayant la plus haute priorité. Il interrompt le thread de priorité inférieure qui était actif. A priorité égale, la spécification du langage précise que l'ordonnanceur choisit un thread à exécuter, en utilisant une file d'attente FIFO[1].

Le thread actif courant s'exécute jusqu'à ce qu'une des conditions suivantes soit vraie :

- un thread de priorité supérieure devient prêt,
- il relâche la CPU, ou la méthode se finit,
- sur les systèmes à temps partagé (décrit ci-dessous), la tranche de temps expire.

Les deux premiers cas privilégient la situation de *threads égoïstes* (*selfish threads*) : entre deux threads de la même priorité, le thread actif courant ne relâche pas la CPU et continue à s'exécuter jusqu'à ce qu'il se finisse naturellement ou qu'il soit interrompu par un thread de plus grande priorité. Certains systèmes, comme Windows 95/NT, ou certaines versions de JDK[2], comme la 1.3, (voir le tableau 12.1), combattent le comportement des threads égoïstes par la stratégie de *temps partagé*.

Le modèle à temps partagé permet la préemption entre les threads d'une même priorité. Ce principe permet à des threads de priorités égales de se passer périodiquement la main les uns les autres. Dans le cadre général, le même principe est appliqué : le thread qui vient de se finir se déplace à la fin de la file de priorité. La présence du mécanisme de temps partagé signifie qu'un compteur de temps interne est déclenché périodiquement, interrompant le thread actif courant et rendant actif

[1]First In First Out = Premier Arrivé Premier Servi
[2]Java Development Kit

| | jdk1.1.8 | | jdk1.2.2 | | jdk1.3 | | jdk1.4 | |
|---|---|---|---|---|---|---|---|---|
| | green | natif | green | natif | green | natif | green | natif |
| Solaris | pas t. part | t. part | pas t. part | t. part | pas supp | t. part | - | - |
| Linux | pas t. part | t. part | pas t. part | t. part | pas supp | t. part | pas supp | t. part |
| Windows | pas supp | t. part | pas supp | t. part | pas supp | t. part | pas supp | t. part |

green/natif = types de threads
t. part = temps partagé
pas supp = pas supporté
- = non spécifié

Tableau 12.1 – Types d'ordonnancement des threads Java, en fonction de modèles

le thread suivant de la file de priorité. Java n'implémente pas ce type d'ordonnancement, mais ne l'interdit pas non plus.

### Modèles d'ordonnancement

Dans différentes implémentations de la machine virtuelle Java, l'ordonnancement des threads Java [OW00] peut être effectué à différents niveaux :
- au niveau de la JVM elle-même - le modèle *threads green*,
- au niveau du système d'exploitation - le modèle *threads natifs*.

**Le modèle threads green.** Les threads green sont des threads au niveau utilisateur et seule la JVM effectue l'ordonnancement, en se basant sur la priorité. Le système d'exploitation n'a pas connaissance des threads Java, la JVM étant pour lui un seul processus et un seul thread. Dans ce modèle, une JVM ne peut exécuter qu'un seul thread à la fois, même sur une machine multi-processeur.

**Le modèle threads natifs.** Dans le modèle de threads natifs, le système d'exploitation, hôte de la JVM, effectue l'ordonnancement des threads.

Deux modèles de programmation existent pour réaliser la correspondance entre les threads utilisateur et les threads du noyau (figure 12.4) :
- le modèle un-vers-un, qui associe chaque thread utilisateur à un thread noyau (par exemple, sous Windows) (partie gauche de la figure),
- le modèle plusieurs-vers-plusieurs, qui répartit plusieurs threads utilisateur sur un nombre de threads noyau (par exemple, sous Solaris) (partie droite de la figure).

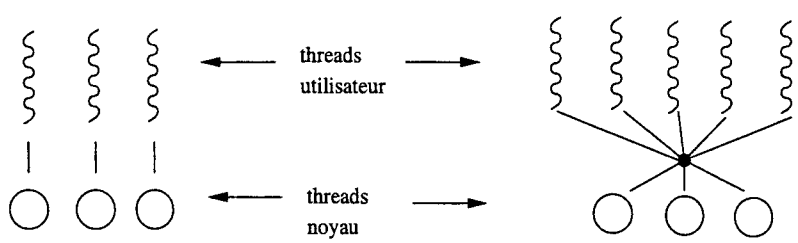



Figure 12.4 – Modèles d'ordonnancement des threads natifs

En fonction du système d'exploitation, différents algorithmes sont utilisés pour l'ordonnancement des threads natifs :

- Sous Windows, il y a une correspondance un à un entre les threads Java et les threads du système d'exploitation (processus). Windows accepte seulement 7 priorités, par rapport au 10 de Java, et aussi 5 classes de priorités. Le système d'exploitation fait la correspondance entre les deux types de priorités, pour ordonnancer les processus suivant le mécanisme préemptif basé sur la priorité.
- Sous Linux, un comportement similaire existe, par la correspondance entre les threads natifs Java et les processus Linux. La différence réside dans les plages de priorités.
- Sous Solaris, il existe un niveau intermédiaire de threads, entre les threads utilisateur et les threads noyau : le processus léger (ou *LWP*). L'ordonnancement des processus en Solaris 2 est basé sur les priorités, en définissant 4 classes d'ordonnancement :
  - la classe temps réel (les processus de cette classe ont la plus haute priorité),
  - la classe système (pour l'exécution des threads noyau),
  - la classe à temps partagé (la priorité des threads de la classe est dynamique, l'algorithme d'ordonnancement étant basé sur la file d'attente multiniveau à retour [SGG01]),
  - la classe interactive (identique à la classe à temps partagé, sauf pour les applications avec fenêtrage, qui ont une priorité plus grande).

  La figure 12.5 montre la correspondance réalisée entre les threads utilisateurs, les processus légers et les threads noyau.

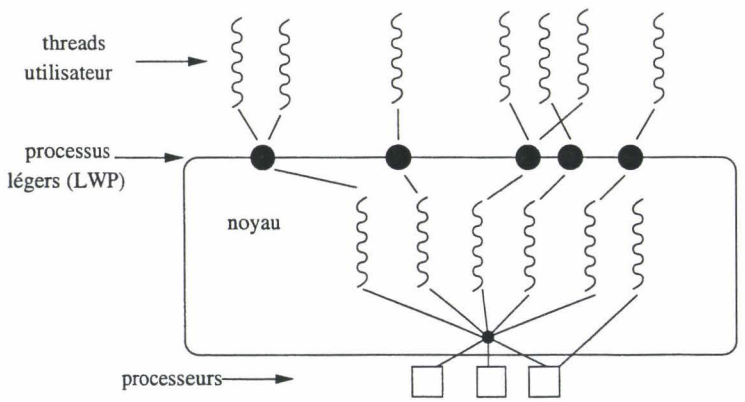



Figure 12.5 – Modèle d'ordonnancement des threads natifs sous Solaris 2

En Java, les deux types de threads existants, natifs et green, imposent un comportement différent lors de l'ordonnancement. Les premiers reposent sur l'ordonnancement du système d'exploitation, alors que les seconds, sont ordonnancés par la JVM.

### 12.1.3 Charge en ADAJ

En ADAJ, le mécanisme d'estimation de charge, proposé en [BOT01], vise à fournir des informations qui indiquent la charge des machines physiques de la plate-forme d'exécution. Ces informations se traduisent par des indicateurs de charge, comparables entre différentes machines. L'objectif du mécanisme était à la fois de mesurer la charge d'une machine physique, et d'être le plus efficace possible, simple et portable.

Le mécanisme de mesure de la charge [BOT01] vise à fournir des informations, appelées *indicateurs de charge*, qui estiment la charge des machines dans la plate-forme d'exécution. Dans

un contexte de système hétérogènes, par rapport à la puissance des machines ou aux politiques d'allocation de la CPU par les ordonnanceurs, l'outil doit être portable.

Le mécanisme de calcul de charge s'appuie sur la mesure du temps moyen passé en attente de la CPU, après l'avoir libérée explicitement (voir la figure 12.6). L'hypothèse d'exploitation de ces informations est que le temps moyen est faible s'il n'y a pas d'autres processus prêts à s'exécuter (ni des processus légers dans la JVM, ni de processus lourds dans la machine support). Le lissage des mesures obtenues s'impose, pour estimer le caractère "prometteur" de la machine physique en termes de disponibilité et afin de ne pas réagir à des fluctuations brusques. Le lissage est réalisé par la moyenne des valeurs de charge se trouvant dans un tampon, pendant une période de temps.

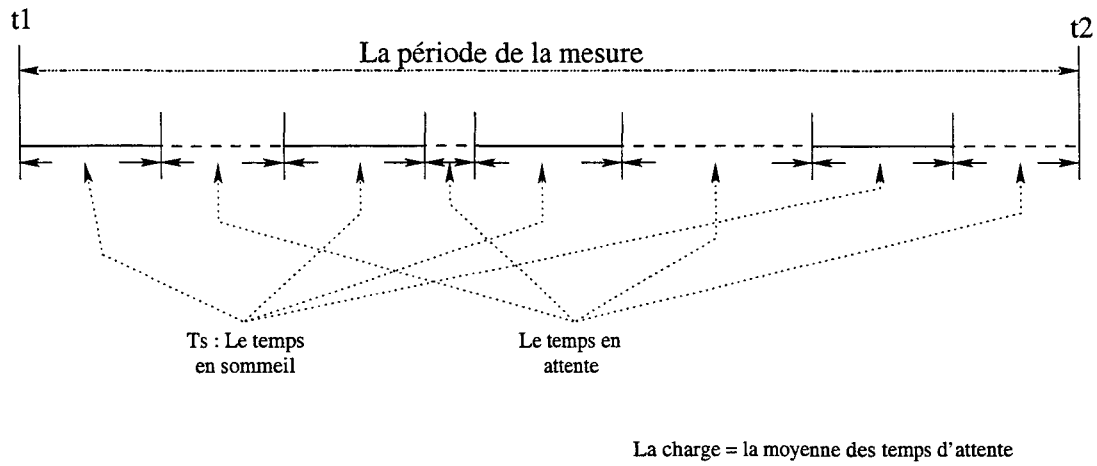



Figure 12.6 – Collecte des informations de charge

La simplicité de l'algorithme permet d'assurer l'obtention de la charge sans un surcoût excessif, qui risquerait de perturber l'exécution des autres applications accueillies dans la machine physique.

Malheureusement, les expérimentations [Bou03] ont montré une dépendance des mesures du type de système d'exploitation et de la politique d'allocation de l'unité centrale aux processus (plus exactement de l'implémentation des threads de la JVM). Le mécanisme aurait dû être corrélé avec un calibrage de la puissance des machines et rendu indépendant de la plate-forme d'exécution, en termes d'ordonnancement de processus.

Ainsi, la charge considérée en ADAJ concerne uniquement l'application ADAJ, qui s'exécute dans un environnement dédié et homogène. La définition de charge sera donnée dans la section 13.4.

## 12.2 Observation de l'évolution de l'application

Le deuxième type d'observation en ADAJ concerne l'évolution de l'application en termes de communication entre les objets. Le mécanisme d'observation est présenté par la suite.

### 12.2.1 Entités observées

L'observation du comportement de l'application complète le premier type d'observation en apportant des informations relatives à l'évolution de l'application. L'évolution d'une application est directement liée aux objets qu'elle contient. Les informations souhaitées concernent : la relation entre les objets, la quantité de traitement lié aux objets.

Une analyse concernant les objets à observer [Bou03] dégage les remarques suivantes :
- l'observation de tous les objets d'une application introduit un surcoût important, vue la quantité d'objets d'une application,
- en ADAJ deux types d'objets se distinguent : les objets locaux et les objets remote. Les objets remote étant les seuls migrables, ils sont intéressants à observer, pour connaître la quantité de travail correspondante qui peut ainsi être déplacée.

Le marquage existant pour les objets remote, en vue de l'observation, classe les objets d'une application ADAJ en :
- *objets globaux* : les objets remote, donc accessibles à distance, migrables, et observables,
- *objets locaux* : les objets Java classiques, non-accessibles à distance, qui ne peuvent pas migrer, et non-observables.

### 12.2.2 Observation des objets globaux

#### Possibilités d'observation des objets globaux

Lors de l'observation d'un objet global, plusieurs informations sont intéressantes à acquérir : la taille de l'objet ou les relations entre les objets (comme : le temps d'exécution d'une méthode, le nombre de méthodes invoquées). La taille de l'objet, corrélée avec le temps d'exécution d'une méthode, donne une information sur la quantité de traitement engendré, qui sera éventuellement déplacée. Les relations entre les objets montrent des *objets fortement communicants* (entre lesquels il y a de nombreuses invocations) à l'opposé des *objets faiblement communicants*. La communication distante est coûteuse. L'intérêt est alors de rapprocher, sur une même machine virtuelle, des objets fortement communicants, situés sur des machines virtuelles différentes.

- Observation de la taille d'un objet
  La taille d'un objet est définie par la taille de toutes les données (donc attributs) qu'il contient. Le mécanisme de calcul de la taille se base sur l'opération de sérialisation. La sérialisation transforme un objet en un flux de données (de manière récursive pour tout attribut), la taille de l'objet étant ainsi la longueur de ce flux.
  Tenant compte que la taille d'un objet ne quantifie pas à 100 % le traitement engendré, une autre information, sur le temps d'exécution d'une méthode, serait utile.
- Observation des relations entre les objets
  Une relation entre des objets se traduit par des invocations. Une invocation d'un objet vers un autre est possible au travers d'une référence. Cette référence est obtenue par :
  - un attribut de l'objet ou
  - le passage de paramètres à une méthode invoquée.

  Plusieurs informations sont intéressantes à considérer lors d'une invocation :
  - le temps d'exécution de la méthode invoquée,
  - le nombre d'instructions bytecode exécutées de la méthode.

  Le temps d'exécution d'une méthode est le temps CPU utilisé par la méthode. Toute invocation sera ainsi observée, en comptant le temps écoulé depuis sa première instruction jusqu'à la dernière.
  Le nombre d'instructions bytecode exécutées d'une méthode est une mesure encore plus fine, de l'importance de la méthode exécutée. Un autre type d'information concerne la fréquence de l'appel d'une méthode, indépendamment du code exécuté par celle-ci.

**Analyse des possibilités et conclusion**

L'observation de la taille d'un objet est intéressante et apporte des informations pertinentes si le traitement engendré est proportionnel, en temps d'exécution, à la quantité de données à traiter. C'est le cas des applications régulières, où cette corrélation est présente. La taille d'un objet est aussi utile lors de la considération des quantités pour la migration : les objets de taille importante prennent un temps plus long lors de la migration, que les objets de petite taille (voir les expérimentations de la section 14.3.3).

L'inconvénient de cette mesure est le surcoût de la sérialisation, dépendant de la taille de l'objet.

L'observation du temps d'exécution d'une méthode est complexe à maîtriser pour les raisons suivantes :

- les moments d'inactivité d'une méthode, dûs aux blocages, utilisateur (à cause d'une suspension du thread d'exécution ou à cause d'une opération d'entrée/sortie) ou système (ordonnancement de processus), sont difficilement quantifiables (il faudrait repérer les appels correspondants dans le programme),
- les appels asynchrones ne sont pas traçables, étant donné que l'appelant récupère toute de suite la main dans la succession d'appels.

L'observation des relations entre les objets peut être réalisée par le comptage des instructions bytecode d'une méthode. Le surcoût est encore plus important que dans le cas précédent, étant donné que chaque instruction bytecode doit être comptée. De plus, il faut que les instructions bytecodes des méthodes appelées, à l'intérieur de la méthode considérée, soient aussi comptées. Le lien entre les méthodes appelées et la méthode considérée doit ainsi être fait.

Une mesure, qui permet d'estimer le volume de communications entre les objets globaux, est le nombre d'invocations de méthodes, approche retenue en ADAJ. Ce type d'information est moins coûteux et moins complexe à implémenter, donnant en même temps une information exploitable. Cela se base sur le fait que toute activité à réaliser dans un système à objets est générée par des invocations de méthodes. Le mécanisme de comptage est décrit dans la section suivante.

### 12.2.3 Comment observer les relations entre les objets au travers de nombre d'invocations de méthodes

L'observation des relations entre les objets au travers d'invocations de méthodes revient à construire dynamiquement le graphe d'objets de l'application, graphe orienté dont les arcs sont étiquetés du nombre d'invocations réalisées. Les objets du graphe seront classés dans deux catégories : globaux et locaux, tandis que les arcs correspondant aux invocations peuvent être :

- d'un objet global vers un objet global,
- d'un objet global vers un objet local,
- d'un objet local vers un objet global.

L'observation d'un objet global, en vue de sa communication avec d'autres objets, comporte :

- l'observation des invocations de l'objet global vers chaque objet global, un autre ou lui-même (relation *OutputGlobalInvocation*, notée *OGI*),
- l'observation des invocations de l'objet global vers tous les objets locaux (relation *OutputLocalInvocation*, notée *OLI*),
- l'observation des invocations d'autres objets (locaux ou globaux) vers l'objet global considéré (relation *InputInvocation*, notée *II*).

La relation *OGI* est définie pour les couples d'objets globaux (étant une relation binaire) tandis que les relations *OLI* et *II* sont définies pour des objets globaux (relations unaires).

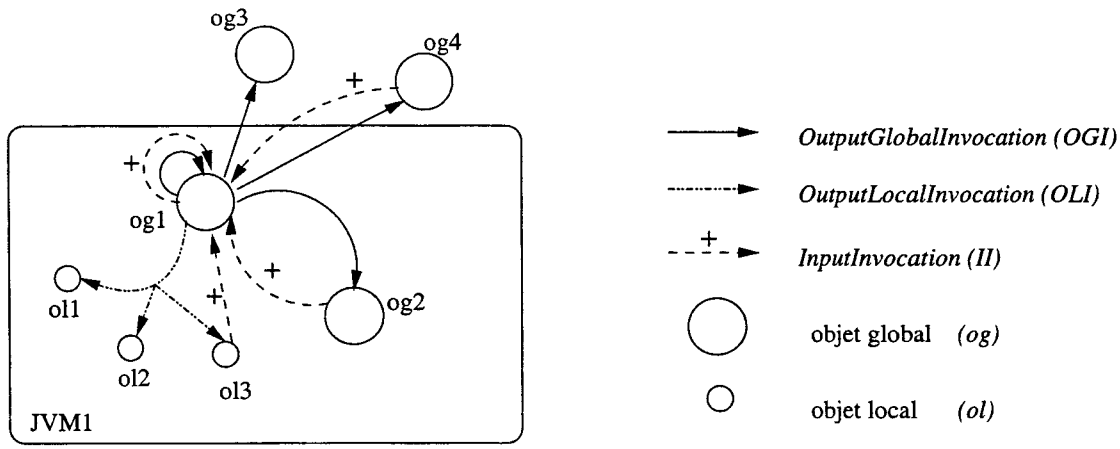



Figure 12.7 – Relations entre les objets d'une application ADAJ

La figure 12.7 montrent les différentes relations de communication qui peuvent exister entre les objets d'une application ADAJ. Entre les objets globaux (d'une même JVM - *og1* et *og2* - ou dans d'autre JVM - *og1* et *og3* ou *og1* et *og4*) il y a la relation *OutputGlobalInvocation*, relation non-symétrique. Ainsi, la valeur de la relation *OGI(og1,og4)* est en général différente de la valeur de la relation *OGI(og4,og1)*. L'objet global *og1* peut avoir des relations avec des objets locaux de la JVM (*ol1*, *ol2* et *ol3*) qui sont quantifiées, par sommation, en relation *OLI*. Tout appel vers un objet global (*og1*, par exemple, dans la figure), venant de l'extérieur, qu'il provienne des objets globaux de la même machine virtuelle ou d'une JVM différente (*og2* respectivement *og4*) ou des objets locaux (*ol3*), est compté dans la relation *II*.

Les invocations considérées pour un objet global peuvent être classées comme :

- *sortantes* : si les invocations proviennent de l'objet global vers les autres objets,
- *entrantes* : si les invocations proviennent d'autres objets globaux ou locaux vers l'objet global considéré.

Les relations se traduisent dans des compteurs attachés à l'objet global, nommés également *OGI*, *OLI* et *II*, correspondant aux relations définies précédemment.

### 12.2.4 Mécanisme global d'observation des relations

L'observation de l'évolution de l'application consiste à observer tous les objets globaux. Cette observation est menée par un composant d'observation (un par JVM), qui maintient à jour les informations concernant les relations de chaque objet global de la JVM. Les informations locales de l'observation sont collectées par un système global, pour la visualisation du graphe d'objets globaux de l'application.

Les compteurs d'observation sont des cumuls d'invocations. Vue la quantité d'invocations réalisées pendant l'exécution d'une application et l'évolution dynamique d'une application, ces cumuls doivent être mis à jour avec une certaine fréquence. Cette opération consiste en un mécanisme de *vieillissement* (*lissage*) de l'information. Le passé récent est intéressant dans l'analyse de l'évolution. Mais, en même temps, si l'analyse est trop réactive par rapport au passé récent, elle risque d'imposer des décisions inadéquates. En supposant que le futur proche a un comportement similaire au passé récent (supposition faite aussi en [CHPB96]), l'information lissée prédit dans le futur les tendances des relations entre les objets d'une application. Le lissage est réalisé à des

intervalles de temps réguliers (donnés par une *fréquence de lissage*), par les opérations suivantes, où *rel* est un compteur correspondant à une des relations *OGI, OLI*, ou *II* et $\alpha$ est une valeur réelle entre 0 et 1 :

- pondérer la valeur précédente lissée avec la valeur courante du compteur,

$$scan_k = \alpha * scan_{k-1} + (1 - \alpha) * rel$$

- remettre à zéro la valeur du compteur,

$$rel = 0.$$

La valeur du paramètre $\alpha$ a été analysée [Bou03], par des expérimentations, pour une application donnée. Son calibrage n'est pas totalement indépendant de l'application, mais reflète le poids que prend le passé récent dans l'évolution de l'application, par rapport au poids du comportement courant.

## 12.3 Conclusions

Ce chapitre décrit des outils nécessaires pour définir la charge d'une machine, information primordiale pour le mécanisme de distribution qui l'utilise.

En ADAJ, l'objectif était de réaliser une distribution à la fois inter et intra application, dans un contexte multi, respectivement mono utilisateur. Cet acquis se basait sur des informations d'observation de l'application d'un côté, et de l'environnement d'exécution, de l'autre. La portabilité réduite du dernier nous a fait considérer, dans un premier temps, uniquement l'équilibrage intra-application, dans un système homogène (le calibrage des puissances des machines n'étant pas réalisé).

L'analyse des différentes informations nécessaires pour l'observation d'une application a montré soit la difficulté de l'extraction de certaines, soit le coût élevé de l'exploitation du mécanisme. Nous avons proposé en ADAJ une observation uniquement du nombre d'invocations de méthodes. L'idée nouvelle du mécanisme se base sur le principe que tout traitement réalisé dans un système à objets se traduit en invocations de méthodes. Et, réciproquement, un nombre important d'invocations génère de l'activité.

Le chapitre suivant montre l'exploitation de l'outil d'observation, pour offrir de l'équilibrage de charge. Le mécanisme d'observation des relations entre les objets sert à définir des critères de choix pour la distribution des objets, à base de fonctions d'attraction ou de répulsion.

# Chapitre 13

# Mécanisme d'équilibrage en ADAJ

## 13.1 Introduction

Dans ce chapitre est présentée la stratégie d'équilibrage de charge en ADAJ.

Le modèle relationnel entre les objets globaux d'une application ADAJ, généré par l'outil d'observation, rappelle ce graphe relationnel construit en [CHPB96] qui permet d'optimiser le placement des objets C++. Dans un autre contexte, d'applications Java, avec des contraintes pour la définition de la charge, ADAJ se base également sur un graphe d'objets construit dynamiquement, à l'exécution, en fonction du nombre d'invocations de méthodes. L'équilibrage de charge revient à redistribuer les objets, donc à modifier la distribution du graphe d'objets sur les machines, tel que chaque machine détient presque la même quantité de calcul, et les communications inter-machines nécessaires à l'échange d'informations soient minimisées. Cette technique est connue sous le nom de *partitionnement de graphes* [SKK99b, SKK99a]. L'adaptabilité de cette technique pour un graphe distribué au départ, comme celui en ADAJ, est difficile, à cause du coût engendré. L'intérêt du mécanisme d'équilibrage en ADAJ n'est pas d'avoir une solution optimale, mais à partir d'heuristiques, de corriger les déséquilibres.

Dans ce mécanisme, deux étapes principales sont identifiées : la détection d'un déséquilibre et sa correction.

Le premier aspect nécessite la *définition de la charge* d'une machine (section 13.4) et des informations sur l'éparpillement des valeurs la mesurant, sur une échelle (section 13.5). La définition de la charge se base sur de *capteurs*, locaux à chaque machine, qui extraient des informations sur l'état des machines et de l'application. Les informations d'éparpillement sont basées sur des métriques statistiques, comme des mesures de dispersion classiques - détection des valeurs anormales - ou des mesures robustes faisant usage des indices de dispersion. Les métriques de base qui mesurent la dispersion sont présentées en section 13.2. Les informations issues des capteurs sont communiquées à un décideur global qui se charge de la détection d'un déséquilibre. La détection du déséquilibre (section 13.6) est essentielle pour la technique d'équilibrage, ce qui nous a amené à considérer différents algorithmes pour la détection. Ces techniques ont été analysées pour estimer leur utilité dans la détection de l'éparpillement des valeurs de charge (section 13.3).

Lors d'un déséquilibre détecté, des *actionneurs*, locaux aux machines, sont informés, pour activer, de manière distribuée, la correction. La correction d'un déséquilibre de charge est réalisée en trois étapes, qui sont les modalités de correction. En considérant les objets globaux comme entités de correction, les problèmes à résoudre sont : quels objets migrer, comment migrer et où migrer ? (section 13.7)

## 13.2 Les mesures de dispersion

Estimer la présence d'un déséquilibre nécessite des informations statistiques sur l'éparpillement des valeurs, sur leur éloignement de leur centre, sur leur variation ou leur dispersion.

Les mesures de dispersion permettent de chiffrer la variabilité des valeurs autour d'un paramètre de position. Pour caractériser l'étalement d'un ensemble de valeurs, les statisticiens ont introduit une série de grandeurs, dont nous allons considérer les principales : l'étendue, l'écart moyen absolu, l'écart type, le coefficient de variation. Pour celles-ci, la définition mathématique et leur interprétation sont données. Considérons, par la suite, l'ensemble de valeurs (appelé aussi *distribution*) $(x_i), i = \overline{1,n}$.

### 13.2.1 L'étendue

*L'étendue* est l'écart entre la plus grande et la plus petite valeur. Cette mesure permet une approche assez simple de la dispersion. L'étendue peut être définie comme la longueur de l'intervalle dans lequel se situent les valeurs de la distribution [BB97]. Par exemple, pour une plage de valeurs entre 10 et 100, l'étendue de la distribution est de 90 unités. De même, pour une plage de valeurs entre 1000 et 1090. Ces résultats font apparaître les limites de l'étendue : elle ne tient pas compte de l'ordre de grandeur des valeurs et est entièrement dépendante des valeurs extrêmes.

La deuxième mesure de dispersion est l'écart par rapport à la moyenne. Deux mesures sont calculées : *l'écart moyen* (considérant la valeur absolue des écarts par rapport à la moyenne) ou *l'écart type* (considérant le carré des écarts par rapport à la moyenne).

### 13.2.2 L'écart moyen (absolu)

Cette mesure consiste à diviser la somme des valeurs absolues des écarts à la moyenne, par le nombre de valeurs de la distribution. L'écart moyen absolu est la distance moyenne à la moyenne.

Formellement, si la moyenne est notée $\overline{x} = \frac{\sum x_i}{n}$, l'écart moyen absolu est $e_m = \frac{\sum_i |x_i - \overline{x}|}{n}$.

Avec cette mesure, les valeurs s'écartent, en moyenne, de $e_m$ de la moyenne. Plus l'écart est élevé, plus la distribution est étendue.

### 13.2.3 L'écart type

Le deuxième indicateur mesurant l'écart par rapport à la moyenne est *l'écart type*. Il consiste à calculer la racine carrée de la moyenne du carré des écarts à la moyenne.

L'écart type se définit comme : $s = \sqrt{\frac{\sum (x_i - \overline{x})^2}{n-1}}$.

Elle se distingue de la première mesure d'écart par le fait qu'elle donne plus de poids aux écarts importants par rapport à la moyenne. Ainsi, cette mesure est sensible aux modifications des valeurs de la distribution et donc aux valeurs extrêmes. L'écart type présente l'avantage d'avoir une signification probabiliste plus intéressante que l'écart moyen. En effet, la théorie des probabilités permet d'estimer la chance qu'a une valeur d'être éloignée de la moyenne de plus d'un certain nombre d'écarts. Chebyshev démontre, dans un théorème [BB97], que pour toute distribution,

- au moins 75 % de valeurs sont dans l'intervalle $(\overline{x} - 2s, \overline{x} + 2s)$,
- au moins 88,9 % de valeurs sont dans l'intervalle $(\overline{x} - 3s, \overline{x} + 3s)$,
- au moins 93,8 % de valeurs sont dans l'intervalle $(\overline{x} - 4s, \overline{x} + 4s)$.

### 13.2.4 Le coefficient de variation

L'écart moyen ou l'écart type donne une mesure de la dispersion de même ordre de grandeur que les valeurs. Pour cela, il n'est pas possible de comparer directement la dispersion de séries exprimée dans des unités différentes, d'où l'impossibilité de généralisation de l'utilisation de la mesure pour différentes distributions. Pour pouvoir faire la comparaison, il faut se ramener à une même unité. C'est le *coefficient de variation.* Il résulte de ce qu'on exprime l'écart type, sous forme d'un pourcentage de la moyenne des valeurs.

Pour $s$ représentant l'écart type et $\overline{x}$ la moyenne, le coefficient de variation est défini comme : $cv = \frac{s}{\overline{x}} * 100$. Ceci permet de comparer la variabilité de deux distributions différentes, parce qu'il n'y a pas d'unité de mesure. Le coefficient de variation est une mesure d'éparpillement des valeurs par rapport à leur moyenne.

Une mesure similaire est l'écart moyen relatif, qui exprime l'écart moyen absolu sous forme d'un pourcentage de la moyenne des valeurs : $e_{mr} = \frac{e_m}{\overline{x}} * 100$.

## 13.3 Problème générique

La détection de l'existence d'un déséquilibre de charge se base sur le problème générique de déséquilibre entre des ensembles de valeurs. Ces valeurs constituent soit une plage homogène, correspondant à un cas d'équilibre, soit une plage hétérogène, caractérisant un déséquilibre. Une plage hétérogène de valeurs implique un éloignement important du centre, c'est-à-dire de la moyenne des valeurs.

Si un déséquilibre est soupçonné, il faut détecter son existence et identifier les valeurs l'engendreant. Ainsi, trois types de valeurs sont identifiés :

- des valeurs normales (notées $n$),
- des valeurs en-dessous des normales (*trop petites* - notées $--$),
- des valeurs en-dessus des normales (*trop grandes* - notées $++$).

Généralement, le mécanisme de détection de ces valeurs implique l'identification de deux seuils : l'un, en deçà duquel les valeurs sont considérées comme étant trop petites, et l'autre, au delà duquel les valeurs sont considérées comme étant trop grandes.

Différents algorithmes ont été analysés pour la détection d'un déséquilibre.

Une première catégorie se construit autour de l'identification des valeurs anormales (tests de normalité - Grubbs [Gru69], Shapiro-Wilk [SW65]). Une analyse de la technique de détection des valeurs aberrantes a montré son inadaptabilité au contexte du problème, à cause de l'existence possible de plusieurs sortes de valeurs aberrantes.

La deuxième catégorie considère des techniques plus robustes qui n'exigent pas l'identification de certaines valeurs comme aberrantes et leur exclusion. Ces statistiques offrent un modèle décrivant la "bonne" partie des valeurs, en se basant sur la moyenne et sur ce qu'il y a autour de cette moyenne, ou, encore, sur l'algorithme de K-Moyennes. Ces algorithmes et leur évaluation sont décrits par la suite.

### 13.3.1 Algorithmes basés sur la moyenne

#### Autour de la moyenne

Les paramètres pour la classification des valeurs sont : la moyenne des valeurs et l'écart moyen absolu.

Les seuils qui déterminent l'intervalle de normalité (dans lequel les valeurs sont considérées normales) sont $\overline{x} - e_m$ et $\overline{x} + e_m$. Formellement, la classification faite est :

- si $x_i < \overline{x} - e_m$, alors la valeur $x_i$ est trop petite,
- si $x_i > \overline{x} + e_m$, alors la valeur $x_i$ est trop grande.

#### K-Moyennes

L'algorithme de K-Moyennes (K-Means, [HW79]) est utilisé dans la classification de $n$ valeurs $x_l$ $(1 \leq l \leq n)$ en groupes, visant deux objectifs :

- minimiser la variabilité en sein d'un même groupe,
- maximiser la distance entre les groupes.

L'algorithme K-Moyennes est itératif, à chaque itération $p$, construisant les groupes $C_1^{[p]}, \cdots, C_k^{[p]}$[1]. L'algorithme suivant présente l'initialisation (l'étape 1), les opérations d'une itération (les étapes 2 et 3) et le cas d'arrêt (l'étape 4) :

étape 1 : initialiser, en choisissant de manière aléatoire $k$ $(1 \leq k \leq n)$ points, de la distribution $x_l$, $(c_1^{[1]}, \cdots, c_k^{[1]})$ qui seront une première série de *centres* pour les $k$ groupes $(C_1^{[1]}, \cdots, C_k^{[1]})$, $c_i^{[1]} = x_l, 1 \leq i \leq k$.

étape 2 : à l'itération $p$, la valeur $x_l$ est associée au groupe $C_i^{[p]}$, si $|x_l - c_i^{[p]}| < |x_l - c_j^{[p]}|, \forall j \neq i$. Cette association met la valeur $x_l$ dans le groupe dont le centre lui est le plus proche.

étape 3 : Pour chaque groupe $C_i^{[p]}$ construit à l'étape 2 pour l'itération $p$, le nouveau centre est déterminé :

$$c_i^{[p+1]} = \frac{1}{n_i} \sum_{x \in C_i^{[p]}} x, \forall i = \overline{1,k},$$

où $n_i$ est le nombre de valeurs dans le groupe $C_i^{[p]}$.

étape 4 : si les centres des groupes à l'itération $p+1$ sont les mêmes que ceux obtenus à l'itération précédente, alors l'algorithme a convergé, sinon l'étape 2 est reprise.

L'algorithme est basé sur deux mesures : la moyenne et l'écart type. La répartition des valeurs dans un groupe est effectuée de façon à minimiser l'indice de dispersion $J$, défini comme :

$$J = \sum_{i=1}^{k} \sum_{x_l \in C_i} |x_l - c_i|^2.$$

L'algorithme de K-Moyennes réalise la répartition des valeurs dans les groupes, tel que $J$ est minimal si le centre de chaque groupe $C_i$ est donné par $c_i = \frac{1}{n_i} \sum_{x_l \in C_i} x_l$.

Les groupes créés dans l'algorithme K-Moyennes sont fortement dépendants des centres initiaux choisis. Par exemple, des groupes différents sont obtenus pour la même distribution (1,242 ; 1,095 ; 1,135 ; 1,165 ; 1,18 ; 1,21 ; 1,232), si les centres initiaux sont 1,095 ; 1,179 ; 1,242 ou respectivement 1,135 ; 1,165 ; 1,242, conformément au tableau suivant :

[1] l'indice du groupe est marqué en indice et le numéro d'itération en exposant

| | centres initiaux | |
|---|---|---|
| | 1,095 ; 1,179 ; 1,242 | 1,135 ; 1,165 ; 1,242 |
| $--$ | {1,095 ; 1,135} | {1,095 ; 1,135} |
| $n$ | {1,165 ; 1,18 ; 1,21} | {1,165 ; 1,18} |
| $++$ | {1,232 ; 1,242} | {1,21 ; 1,232 ; 1,242} |

Dans la classification nécessaire pour la détection d'un déséquilibre, trois groupes doivent être identifiés. Ainsi, $k = 3$ et les centres initiaux considérés sont : la valeur minimale, la valeur maximale et la moyenne. Par rapport à l'algorithme précédent, la définition des seuils est faite lors de la détection des groupes, car ceux-ci définissent les plus petites, les plus grandes valeurs et les valeurs normales.

L'algorithme de K-Moyennes appliqué aux différentes distributions de valeurs, pour le cas particulier de $k = 3$, et les centres initiaux respectivement {17 ; 17,7 ; 18}, {1,095 ; 1,179 ; 1,242} et {1 ; 12,64 ; 36,53}, forme les groupes suivants :

| valeurs | distr 1 | distr 2 | distr 3 |
|---|---|---|---|
| $x_1$ | 17 | 1,242 | 1 |
| $x_2$ | 18 | 1,095 | 4 |
| $x_3$ | 18 | 1,135 | 7 |
| $x_4$ | 18 | 1,165 | 10 |
| $x_5$ | 18 | 1,18 | 13 |
| $x_6$ | 18 | 1,21 | 17 |
| $x_7$ | 17 | 1,232 | 36,53 |
| $--$ | {17 ; 17} | {1,095 ; 1,135} | {1 ; 4 ; 7} |
| $n$ | {} | {1,165 ; 1,18 ; 1,21} | {10 ; 13 ; 17} |
| $++$ | {18 ; 18 ; 18 ; 18 ; 18} | {1,232 ; 1,242} | {36,53} |

Les deux algorithmes présentés détectent toujours des valeurs trop petites ou/et trop grandes, lors de la considération des valeurs proches, sauf pour le cas d'une égalité totale entre les valeurs.

### K-Moyennes combiné

Le problème apparu pour les deux algorithmes précédents (autour de la moyenne et K-Moyennes) dérive du fait que les petites variations de valeurs par rapport à la moyenne ne sont pas considérées comme non significatives. Par exemple, une fluctuation de 10 % entre les valeurs ne devrait pas être considérée comme importante, donc engendrer la détection d'un déséquilibre.

Le moyen d'estimer si ces fluctuations sont importantes est de considérer le rapport entre l'écart type et la moyenne. Ce rapport caractérise le coefficient de variation et il est indépendant des valeurs considérées. Ainsi, l'algorithme K-Moyennes combiné se constitue en deux phases :

- détecter si les valeurs sont dans une plage hétérogène (càd si un déséquilibre existe),
- identifier les valeurs qui engendrent le déséquilibre, si ce dernier est détecté.

Les algorithmes précédents réalisent les deux phases en une seule, d'où la détection quasi-permanente d'un déséquilibre.

Pour la détection d'un déséquilibre, le coefficient de variation est comparé avec un seuil fixé, *seuil* :

- si $cv \leq seuil$, il n'existe pas de déséquilibre,
- si $cv > seuil$, le déséquilibre est détecté et l'algorithme K-Moyennes forme les groupes avec les valeurs trop petites, normales et trop grandes.

Un exemple pour l'application de l'algorithme de K-Moyennes combiné est illustré dans le tableau suivant, pour des ensembles de 7 valeurs, en considérant un seuil de 30 %.

| valeurs | distr 1 | distr 2 | distr 3 |
|---|---|---|---|
| $x_1$ | 17 | 1,242 | 1 |
| $x_2$ | 18 | 1,095 | 4 |
| $x_3$ | 18 | 1,135 | 7 |
| $x_4$ | 18 | 1,165 | 10 |
| $x_5$ | 18 | 1,18 | 13 |
| $x_6$ | 18 | 1,21 | 17 |
| $x_7$ | 17 | 1,232 | 36,53 |
| $\overline{x}$ | 17,7 | 1,179 | 12,64 |
| $s$ | 0,487 | 0,053 | 11,82 |
| $cv$ | 2,7 % | 4,5 % | 93,54 % |
| déséquilibre | non | non | oui |

Déterminer le seuil reste un problème, mais l'avantage est son indépendance vis-à-vis des valeurs considérées.

### 13.3.2 Analyse, évaluation des algorithmes et commentaires

Une mesure considérée pour estimer la dispersion des valeurs est l'écart par rapport à la moyenne. Cet écart est la différence entre une valeur et la moyenne de la distribution. Comme la somme des écarts à la moyenne est égale à zéro, la somme des valeurs absolues des écarts calcule l'écart moyen et la somme des carrés des écarts calcule l'écart type. Les deux mesures se distinguent par le poids donné aux écarts importants (la valeur absolue, dans le cas de l'écart moyen versus la valeur du carré dans le cas de l'écart type). Prendre le carré des écarts à la moyenne c'est renforcer le poids des valeurs extrêmes. Par exemple, pour des distributions avec la même valeur de la moyenne, la variation de l'écart moyen et de l'écart type est la suivante :

| valeurs | distr 1 | distr 2 | distr 3 | distr 4 |
|---|---|---|---|---|
| $x_1$ | 5 | 1 | 5 | 1 |
| $x_2$ | 9 | 10 | 3 | 2 |
| $x_3$ | 10 | 10 | 6 | 1 |
| $x_4$ | 10 | 10 | 1 | 1 |
| $x_5$ | 1 | 4 | 20 | 30 |
| $\overline{x}$ | 7 | 7 | 7 | 7 |
| $e_m$ | 3,2 | 3,6 | 5,2 | 9,2 |
| $s$ | 3,94 | 4,24 | 7,52 | 12,86 |

L'algorithme autour de la moyenne considère que les valeurs normales sont autour de la moyenne. Il évalue une répartition par un pourcentage. C'est le pourcentage des valeurs qui se trouvent dans l'intervalle centré sur la moyenne et dont le rayon (la demi longueur) est un certain nombre de fois l'écart type. Ainsi, le pourcentage du nombre de valeurs dans l'intervalle $(\overline{x} - 2s, \overline{x} + 2s)$ est d'au moins 75 %. Pour un intervalle de normalité autour de la moyenne, de rayon $e_m$, rien n'est dit sur le pourcentage des valeurs retrouvées. Tenant compte de la corrélation remarquée précédemment entre l'écart moyen et l'écart type, un comportement similaire est attendu. La considération de l'intervalle de normalité (entre $\overline{x} - e_m$ et $\overline{x} + e_m$) n'assure pas que des valeurs proches de la limite inférieure de l'intervalle seront considérées comme trop petites ou, à l'opposé, des valeurs proches de la limite supérieure de l'intervalle seront considérées comme trop grandes. Cet algorithme risque de ne pas les détecter. Par exemple, pour la distribution (0 ; 1 ; 2 ; 12 ; 13 ; 14), l'intervalle de normalité est entre 1 et 13, et seules les valeurs 0 et 14 sont détectées extrêmes. Dans certains cas, il serait intéressant de distinguer directement 2 groupes de valeurs extrêmes : {0 ; 1 ; 2} et {12 ; 13 ; 14}.

L'algorithme de K-Moyennes présente une solution au problème précédent, identifiant directement les deux groupes. L'inconvénient qu'il présente est de considérer des variations mineures comme importantes. D'où, le coefficient de variation introduit, dans l'algorithme de K-Moyennes combiné, qui permet de ne pas tenir compte de l'ordre de grandeur des valeurs. Pourtant, il faut remarquer que le coefficient de variation est très sensible à la valeur moyenne. Par exemple, pour deux distributions ayant la même variabilité, (1 ; 2 ; 3 ; 4) et (21 ; 22 ; 23 ; 24), le coefficient de variation est, d'un côté, 52 %, de l'autre, 5 %. Il n'est pas possible de donner une règle générale pour savoir à partir de quel niveau un coefficient de variation est acceptable. Il faudra une détermination du seuil expérimentale.

## 13.4 Définition de la charge d'une JVM

L'équilibrage de charge pour l'exécution d'une application Java distribuée suppose d'équilibrer les différentes charges des JVM, dans le contexte d'une variation de charge à l'intérieur de l'application et du partage des ressources d'une machine avec d'autres processus, qui s'exécutent sur la machine. Le mécanisme d'équilibrage peut intervenir seulement à l'intérieur de l'application, par une redistribution de la charge, visant ainsi une meilleure performance, en temps d'exécution de l'application. Ainsi, une caractérisation de la charge des JVM de l'environnement distribué doit être réalisée, conformément à sa définition dans la section 12.1.1. Vues les contraintes liées à l'obtention et le calibrage de la charge d'un système hétérogène et multi-utilisateurs, l'approche présentée par la suite se focalise uniquement sur un équilibrage intra-application, pour des applications s'exécutant dans un système homogène, dédié.

Dans une machine virtuelle Java, les entités de l'application sont les objets. Quand les objets existent et des méthodes sont invoquées sur eux, la JVM crée une charge. Dans le cas contraire, soit à cause de l'absence d'objets, soit à cause du blocage des méthodes, la JVM ne consomme pas de CPU pour l'application.

L'utilité des threads sur une JVM intervient lors de la présence de plusieurs tâches à exécuter, dont certaines peuvent être bloquées. C'est typiquement le cas des fragments ADAJ, entités multithreadées et des appels asynchrones ou de type distribute qui s'exécutent sur eux. L'attente d'un résultat non encore disponible, ou, en général, des communications synchrones, bloquent le thread qui exécute l'appel. Avoir plusieurs threads par JVM permet d'utiliser la CPU au maximum, en donnant le temps affecté à un thread qui se bloque à un autre thread prêt à s'exécuter. Ainsi, la charge d'une JVM est directement liée au nombre de threads s'exécutant dans la JVM. Cette métrique correspond à la longueur de la file de tâches en attente d'exécution dans un système à processus.

Java permet, grâce à une méthode de la bibliothèque[2], de déterminer le nombre de threads s'exécutant sur la JVM. Ce nombre compte aussi des threads internes de la machine virtuelle, liés au ramasse-miettes ou à la gestion de la machine, mais leur prise en compte est judicieuse parce que l'application, s'exécutant dans la JVM, est ralentie par l'activation de ces threads.

L'inconvénient de l'outil Java est qu'il ne fait pas la différence entre des threads bloqués (*non activables*) et les threads prêts à s'exécuter (*activables*). Ainsi, des threads bloqués existant sur une JVM sont considérés comme charge potentielle, mais en réalité, il n'y a pas de CPU utilisé pour l'application. Une nouvelle mesure est introduite pour différencier entre les threads activables et non activables : la quantité de travail de la JVM.

La quantité de travail de la JVM est définie comme la somme des quantités de travail de chaque

[2]la méthode static int Thread.activeCount()

objet global. Un objet global réalise un travail proportionnel au nombre d'invocations entrantes (*II* et *OGI* vers lui-même) et le nombre d'invocations qu'il réalise vers les objets locaux. Les objets locaux ne sont pas observés et ne migrent pas, mais les communications d'un objet global vers les objets locaux, qui lui sont liés, sont considérées dans la quantité de travail de l'objet global. Lors de son éventuelle migration, si les communications avec les objets locaux persistent, elles seront transférées sur la nouvelle localisation de l'objet global, donc elles feront partie de la quantité de travail de la nouvelle machine accueillant l'objet global.

Formellement, la quantité de travail d'un objet *obj* est $WP_{obj}$, définie de la manière suivante :

$$WP_{obj} = OGI(obj, obj) + II(obj) + OLI(obj)$$

et la quantité de travail de la machine virtuelle $JVM_i$ est

$$WP_i = \sum_{obj \in JVM_i} WP_{obj}.$$

## 13.5 Définition du déséquilibre

La métrique de charge en ADAJ est définie par le nombre de threads et la quantité de travail. Sans être quantifiable, la métrique peut classer les machines en sous-chargées, sur-chargées et normalement chargées. Ce principe de classification repose sur la distinction entre les charges des machines à partir du nombre de ses threads et de la prise en compte de la quantité de travail qu'ils représentent.

Tenons compte de l'observation faite dans la section précédente, à savoir : le nombre de threads d'une machine quantifie généralement le travail à faire dans la JVM de façon équivalente à la longueur de la queue CPU dans les systèmes à base de processus. Cette mesure n'est pas suffisante. En affinant, la quantité de travail considérée donne la classification suivante (voir aussi la figure 13.1) :

- si le nombre de threads est important, et la quantité de travail est grande, la machine est sur-chargée parce qu'il y a potentiellement beaucoup de travail,
- si le nombre de threads est petit, la machine est sous-chargée, parce qu'il n'y a pas potentiellement, beaucoup de travail dans la JVM,
- si, quelque soit le nombre de threads, la quantité de travail est petite, la machine est sous-chargée. Souvent ces threads sont en attente de résultats ou bloqués dans une opération d'entrée/sortie.

L'importance d'une mesure (nombre de threads ou quantité de travail) est toujours relativisée par rapport aux autres mesures et décidée en fonction des mesures de dispersion.

Une sous-classification peut être réalisée entre les machines avec un petit nombre de threads, en ordre croissant de la valeur de la quantité de travail, ou entre les machines avec des petites quantités de travail, en ordre croissant du nombre de threads. Pourtant, rien ne peut être inféré sur une classification entre une machine avec beaucoup de threads et petite quantité de travail et une machine avec une grande quantité de travail et peu de threads.

## 13.6 Détection du déséquilibre

L'algorithme pour la détection d'un déséquilibre est essentiel pour le mécanisme d'équilibrage de charge. En fonction de la décision prise, la charge sera redistribuée ou non. Plusieurs algorithmes ont été testés pour classer les machines virtuelles en fonction de leur charge. Un premier algorithme

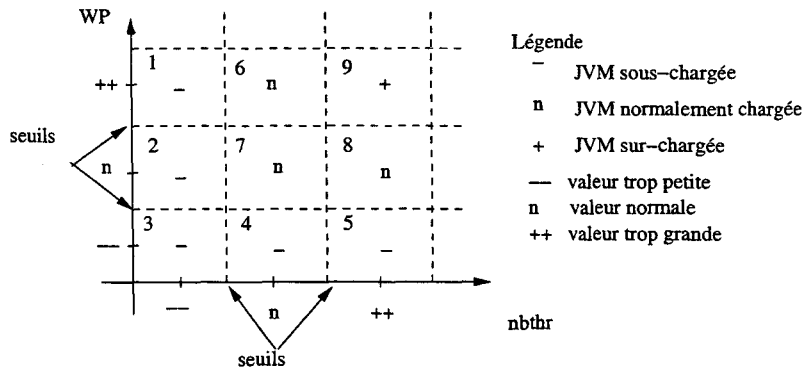



Figure 13.1 – Classification des JVM en fonction des métriques de charge (les zones sont numérotées de 1 à 9)

se base sur la moyenne, prenant l'intervalle de la normalité autour de la moyenne, tandis que les 3 autres se basent sur la méthode de classification de K-Moyennes. Tous les algorithmes sont décrits ci-dessous.

### 13.6.1 Algorithme autour de la moyenne

L'algorithme autour de la moyenne considère, pour les deux métriques de charge (le nombre de threads et la quantité de travail), qu'un intervalle de normalité se situe autour de la moyenne de la façon suivante :

- pour le nombre de threads, entre 3[3] et $nbthr_{moy} + nbthr_{ecartmoy}$,
- pour la quantité de travail, entre $WP_{moy} - WP_{ecartmoy}$ et $WP_{moy} + WP_{ecartmoy}$,

où $nbthr_{moy} = \frac{\sum_{i=1}^{n} nbthr_i}{n}$ et $nbthr_{ecartmoy} = \frac{\sum_{i=1}^{n} |nbthr_i - nbthr_{moy}|}{n}$ . De même pour la quantité de travail $WP$.

La limite inférieure 3 du nombre de threads de l'intervalle de normalité coïncide avec le cas de zéro thread utilisateur. Cet intervalle réduit fortement le cas où les valeurs sont trop petites, ce qui coïncide avec l'inexistence de traitement.

Ainsi, l'algorithme de classification et le suivant, où $i$ est l'indice d'une JVM :

- si $(nbthr_i <= 3$ ou $WP_i < WP_{moy} - WP_{ecartmoy})$ alors
  la machine $JVM_i$ est sous-chargée,
- si $(nbthr_i > nbthr_{moy} + nbthr_{ecartmoy}$ et $WP_i > WP_{moy} + WP_{ecartmoy})$ alors
  la machine $JVM_i$ sur-chargée.

### 13.6.2 Algorithmes basés sur la méthode K-Moyennes

Les algorithmes basés sur la méthode K-Moyennes classent les machines en trois groupes, en fonction des métriques de charge, autour de la moyenne, de la plus petite valeur et de la plus grande valeur. Un transfert de charge ne peut être fait que dans le cas où il existe des valeurs extrêmes trop grandes, pour les métriques de charge, et, en face, des valeurs extrêmes trop petites ; on pourra alors les "coupler en destinataire-source". Ainsi, s'il n'existe pas de déséquilibre entre le nombre de threads et entre les quantités de travail, il n'y a pas de déséquilibre dans le système. Dans cette hypothèse, il y a potentiellement une redistribution de charge seulement s'il existe un déséquilibre

[3]ces trois threads font partie de l'environnement d'exécution standard

pour les deux métriques. De plus, l'algorithme de détection de déséquilibre comprend deux phases : une première phase détecte si un déséquilibre existe, et la deuxième phase détecte quelles sont les valeurs qui engendrent le déséquilibre, ou quelles sont les machines sur et sous-chargées.

**Algorithme 30** Le déséquilibre est détecté par la comparaison de deux ratios avec un seuil donné. Les deux ratios sont :

- le rapport entre l'écart moyen du nombre de threads et le nombre moyen de threads, $nbthr_{ecrtmoy}/nbthr_{moy}$,
- le rapport entre l'écart moyen de la quantité du travail et la quantité moyenne de travail, $WP_{ecrtmoy}/WP_{moy}$.

Le seuil rend acceptable l'écart entre la plus grande et la plus petite valeur. Plus le seuil est petit, plus l'étendue acceptée est petite.

L'algorithme de classification est le suivant :

- si ($nbthr_{ecrtmoy}/nbthr_{moy} > seuil$ et $WP_{ecrtmoy}/WP_{moy} > seuil$) alors
  générer, à l'aide de l'algorithme de K-Moyennes, les 3 groupes pour les valeurs correspondant au nombre de threads ($Sous_{thr}$, $N_{thr}$ et $Sur_{thr}$) et à la quantité de travail ($Sous_{WP}$, $N_{WP}$ et $Sur_{WP}$), et conclure :
  - si $JVM_i \in Sous_{thr}$ ou $JVM_i \in Sous_{WP}$ alors $JVM_i$ est sous-chargée,
  - si ($JVM_i \in Sur_{thr}$ et $JVM_i \in Sur_{WP}$) alors $JVM_i$ est sur-chargée.

**Algorithme 40** L'algorithme précédent prend des mesures instantanées du nombre de threads. L'expérimentation a montré que cette métrique renvoie des valeurs qui ne correspondent pas toujours à l'évolution de la JVM. Des threads de la JVM, comme ceux liés au ramasse-miettes, sont aussi comptés. Pour éviter de réagir trop vite à de telles fluctuations, la valeur du nombre de threads est lissée, à 50 %, avec la valeur précédente. La même opération est réalisée pour la quantité de travail, pour conserver le même comportement.

**Algorithme 50** Une des mesures de dispersion dans la littérature est le coefficient de variation. Les algorithmes précédents utilisent une mesure relativement proche, basée sur l'écart moyen absolu. La variance utilise plutôt l'écart type. Ainsi, un autre algorithme est proposé qui considère les deux coefficients de variation :

- le coefficient de variation du nombre de threads, $nbthr_{ecrttype}/nbthr_{moy}$,
- le coefficient de variation de la quantité de travail, $WP_{ecrttype}/WP_{moy}$.

L'algorithme de classification reste le même que le précédent.

## 13.7 Correction du déséquilibre

Le déséquilibre de charge détecté est corrigé par un déplacement de charge d'une machine sur-chargée vers une machine sous-chargée. Tenant compte que la charge est générée par l'exécution de méthodes sur un objet de la machine sur-chargée, la migration de l'objet sur une machine sous-chargée engendre également un déplacement de la charge correspondante, générée par l'objet, sur la machine sous-chargée. Ainsi, équilibrer les charges des JVM revient à redistribuer judicieusement les objets. Cette redistribution ne s'applique pas à tous les objets, mais seulement à ceux qui pourraient équilibrer la charge par migration. Donc, seuls les objets globaux se trouvant sur des machines sur-chargées sont analysés pour qu'un (ou plusieurs) soit (soient) déplacé(s) sur une machine sous-chargée, en déplaçant de la charge. Les questions soulevées par la redistribution des

objets sont : quoi, où, comment et quand migrer ? Ces questions trouvent les réponses dans la politique de sélection des candidats, la politique de localisation et le moyen de transfert (présentés de manière générale dans le chapitre 10).

### 13.7.1 La politique de sélection

L'entité à migrer est l'objet global qui, par sa migration, peut corriger un déséquilibre de charge. Parmi les objets globaux d'une JVM, certains ont des caractéristiques plus intéressantes pour être migrés. Ces caractéristiques sont liées aux relations avec les autres objets de la machine et à la quantité de charge transportée. Deux relations interviennent :

- *l'attraction d'un objet global vers la JVM* et
- *le poids de l'objet global.*

L'attraction d'un objet global vers la JVM est quantifiée en termes de communications, par les liaisons qu'il a avec les autres objets globaux de la machine. Une forte attraction implique des communications fréquentes, qui seront réalisées à distance lors de la migration de l'objet. Une faible attraction lui permettra de quitter la machine courante, pour une autre, sans engendrer des communications distantes importantes. Ainsi, moins l'objet est attiré par la JVM courante, plus il est intéressant à être migré.

Le poids, d'un objet global à migrer, donne la charge à enlever de la machine courante. Ce poids est quantifié par la quantité de travail de l'objet global. Un objet, dont la quantité de travail est importante, montre une activité continue, d'où des difficultés à prévoir pour sa migration. De plus, dans une comparaison des stratégies d'équilibrage de charge dynamique, [CLZ98] constate qu'une granularité importante de l'objet à migrer risque de renverser le rôle des machines impliquées (la source deviendra une destination et vice-versa). A l'opposé, un objet avec une faible quantité de travail n'apporte pas de changement significatif dans la charge, par sa migration. De plus, il y a le risque que le coût de migration ne soit pas absorbé par la nouvelle répartition de charge générée. En conclusion, la décision est de déplacer un objet dont la quantité de travail ne soit pas trop importante, ni trop petite. Donc, plus la distance par rapport à la quantité moyenne de travail des objets est petite, plus l'objet est intéressant pour une migration.

Formellement, les deux relations considérées sont :

- l'attraction de l'objet global *obj* vers la JVM courante, *attr*,

$$attr(obj) = \sum_{o \in JVM} (OGI(obj, o) + OGI(obj, o)),$$

- la distance par rapport à la quantité moyenne de travail de l'objet *obj*,

$$dist_{m_{WP}}(obj) = |WP_{obj} - m_{WP}|,$$

où $m_{WP} = \frac{\sum_{obj \in JVM} WP_{obj}}{n}$ ($n$ est le nombre d'objets globaux sur la JVM), et $WP_{obj} = OGI(obj, obj) + II(obj) + OLI(obj)$.

Les critères de choix doivent être satisfaits simultanément ; donc, pour cela, une fonction d'agrégation est utilisée. L'agrégation exige une même unité de mesure pour les valeurs considérées. Pour relativiser les deux valeurs pour un objet global *obj*, des relations du type pourcentage sont utilisées :

- le pourcentage de l'attraction de l'objet global *attr*,

$$\%attr(obj) = \frac{attr(obj)}{\sum_{obj \in JVM} attr(obj)},$$

- le pourcentage de la distance par rapport à la quantité moyenne de travail de l'objet,

$$\%dist_{mWP}(obj) = \frac{dist_{mWP}(obj)}{\sum_{obj \in JVM} dist_{mWP}(obj)}.$$

Amenées à une même unité, les relations interviennent avec un poids différent dans la somme pondérée

$$clsmt(obj) = \alpha_{attr} * \%attr(obj) + (1 - \alpha_{attr}) * \%dist_{mWP}(obj),$$

où $\alpha_{attr}$ est un réel entre 0 et 1. L'objet global le mieux classé par rapport à la relation $clsmt$, est choisi pour migrer.

Le choix du coefficient $\alpha_{attr}$ reste expérimentale, indiquant que plus $\alpha_{attr}$ est grand, plus le poids donné à l'attraction de l'objet est grand, et plus $\alpha_{attr}$ est petit, plus l'importance du travail de l'objet à migrer est grande.

### 13.7.2 La politique de localisation

Les objets globaux allègent la charge d'une JVM sur-chargée, par leur migration. La question qui se pose est : où migreront ces objets ? Evidemment, la destination potentielle est une machine sous-chargée, dont l'existence est certaine, par la détection du déséquilibre. Parmi ces machines, certaines présentent plus d'avantages que d'autres, de point de vue quantité de communications avec l'objet à migrer ou quantité de travail.

Les communications de l'objet à migrer avec les objets globaux situés sur d'autres JVM traduisent une attraction externe de l'objet. Une relation d'attraction de l'objet global $obj$ vers la $JVM_i$ est définie comme :

$$attrext_i = \sum_{objext \in JVM_i} OGI(objext, obj) + OGI(obj, objext).$$

Plus l'objet est attiré vers l'extérieur par une machine sous-chargée, plus il est intéressant que sa destination soit la machine correspondante.

En même temps, il peut y avoir la situation de deux machines sous-chargées vers lesquelles l'objet est attiré avec la même intensité. Dans ce cas, la machine la moins chargée des deux, en terme de quantité de travail de la JVM, sera préférée.

Ces deux remarques font transparaître deux contraintes à satisfaire par la machine destinataire. Il faut que :

- l'attraction externe de l'objet soit maximale vers la machine,
- la quantité de travail de la machine soit minimale.

Pour la même raison que précédemment, dans le but de satisfaire deux conditions en même temps, la somme pondérée de deux valeurs doit être considérée. Ayant des unités différentes, les valeurs seront relativisées à la somme de toutes les valeurs, correspondant à chaque relation :

- $\%attrext_i = \frac{attrext_i}{\sum attrext_i}$,
- $\%WP_i = \frac{WP_i}{\sum_i WP_i}$

La fonction d'agrégation considère les deux valeurs, mais dans un sens différent, vu que les critères sont opposés. La somme pondérée $decision_i = \alpha_{attre} * \%attrext_i - (1 - \alpha_{attre}) * \%WP_i$, avec $\alpha_{attre} \in [0, 1]$ permet de calculer en même temps la chance que l'objet a, de migrer sur la machine $JVM_i$. La machine qui maximise cette somme sera choisie comme nouvelle localisation de l'objet.

Le choix du coefficient $\alpha_{attre}$ est également expérimentale, et en augmentant la valeur, l'accent sera mis sur l'optimisation des communications distantes, plutôt que sur le choix de la moins chargée des machines sous-chargées.

L'avantage de l'outil d'observation est la disponibilité d'une information plus riche qui permet de prendre en compte, simultanément, lors de la politique de localisation, le déséquilibre de charge et la minimisation des communications.

### 13.7.3 Le moyen de transfert

Le mécanisme utilisé pour la migration d'un objet se base sur la sérialisation. La technique de migration, reprise de JavaParty, impose certaines contraintes pour que la migration soit réussie. La contrainte de la mise en œuvre est liée à l'impossibilité, en Java, d'avoir accès à la pile d'exécution. Ainsi, la migration en JavaParty ne peut pas être effectuée pendant qu'une méthode s'exécute sur l'objet à migrer. L'avantage qui en résulte est la facilité de la mise en œuvre : lors de la migration d'un objet, il n'existe pas de méthodes en cours d'exécution et lors de l'arrivée de l'objet à sa nouvelle localisation, les méthodes invoquées s'exécutent sur l'objet migré. Le mécanisme détaillé de la migration en JavaParty, ainsi que l'évaluation de son coût sont décrits respectivement dans les sections 3.3.3 et 14.3.3.

Pour des méthodes ayant un temps d'exécution important, la contrainte de migration JavaParty risque d'être importante. Pour débloquer des objets se trouvant dans cette situation, la migration en ADAJ offre la possibilité de migration des threads, plus exactement des objets avec une méthode en cours d'exécution. La migration de ces objets est appelée *migration d'objets actifs*, JavaParty proposant la *migration des objets passifs*. Ce procédé est basé sur la transformation de bytecode, à l'opposé du procédé choisi par le projet SIRAC [Bou99, Bou01], qui utilise l'extension de la machine virtuelle Java. Notre procédé fait appel au mécanisme de JavaParty, ce qui fait que la migration en ADAJ apporte un surcoût par rapport à la migration en JavaParty. La technique de migration ADAJ et l'estimation de son surcoût sont présentées dans le chapitre suivant. Le procédé pourrait être étendu pour le cas de plusieurs méthodes s'exécutant sur l'objet, mais en augmentant le surcoût.

Lors de la redistribution des objets, exigée par le procédé d'équilibrage de charge, une technique de migration est employée. Le choix entre les deux techniques de migration, en JavaParty et en ADAJ, devra être fait. La première présente l'avantage d'être simple, donc moins coûteuse que la deuxième qui est plus complexe, mais plus générale, permettant la migration des objets actifs. Ces avantages et inconvénients imposent un compromis entre le coût de migration et la réussite de la migration. Deux types de transfert ont été proposés. Le premier est basé uniquement sur la migration JavaParty. Les objets, candidats à la migration, sont essayés un par un, dans l'ordre donné par la classification, et le procédé est arrêté à la première tentative réussie. Le deuxième type utilise les deux types de transfert, JavaParty et ADAJ. Un seul objet est essayé avec le mécanisme JavaParty ; si le transfert échoue, le même objet est essayé avec le mécanisme ADAJ.

### 13.7.4 Quand migrer ?

Les moments de migration apparaissent lors de la prise de décision locale sur la politique de sélection et de localisation. Ces décisions sont corrélées avec l'information centrale reçue par une machine sur-chargée. La migration est ainsi dépendante de la décision prise par la politique de décision sur un déséquilibre. La présence d'un déséquilibre est testée à des intervalles de temps réguliers. Ce sont les *moments d'inspections*. En ADAJ, la fréquence d'inspection est fixe, et pas adaptative comme en [MDLT96]. L'introduction de l'adaptabilité de la fréquence d'inspection, dans

cet article, aidait à trouver un compromis entre un maintient d'une information de charge à jour du système, et un surcoût de calcul et communication minimal. Le maintien d'un état de charge récent dans le système est réalisé par la variation de la fréquence des échanges d'informations. En ADAJ, une estimation est donnée par l'intermédiaire de la méthode de lissage de valeurs qui prend en compte aussi bien le passé récent que le présent.

## 13.8 Dispositif général

Le dispositif général d'équilibrage de charge, en ADAJ, est constitué de trois composants :
- *le composant d'observation*, qui extraît les informations liées à l'évolution de l'application (mécanisme présenté dans la section 12.2), et les transforme en métriques de charge suivant l'algorithme présenté en section 13.5,
- *le composant de décision*, qui, à partir d'une connaissance globale du système, constate et confirme, l'existence d'un déséquilibre, en appliquant la politique de décision,
- *le composant de correction*, qui décide sur la quantité de charge à déplacer et sa destination, en appliquant la politique de sélection et de localisation.

Les différents composants interagissent (conformément à la figure 13.2) afin de mieux équilibrer les charges dans le système.

La politique d'information est périodique, à l'initiative du composant central, qui collecte, à des intervalles de temps réguliers, les informations de charge contenues dans chaque composant d'observation. L'information transmise contient le nombre de threads et la valeur de la quantité de travail de la JVM. Le nombre de threads est une valeur instantanée (brute), tandis que la quantité de travail de la JVM est calculée en fonction des valeurs lissées des compteurs d'observation, pendant une période de temps. Ces informations sont analysées par le composant de décision, pour décider si une redistribution des objets est nécessaire. La politique de décision appliquée classe les machines en sur-chargées, sous-chargées ou normalement chargées, et de manière asynchrone, toutes les machines sur-chargées sont informées sur leur état, réceptionnant l'ensemble de machines sous-chargées. Cette réception est réalisée localement, dans chaque JVM sur-chargée, par le composant de correction, qui, de manière indépendante, décidera de la charge à déplacer. La nouvelle localisation des objets sera une des (plusieurs) machines sous-chargées. Pour éviter la congestion d'une machine sous-chargée, effet constaté dans les expérimentations (voir la section 15.4.1), indépendamment des critères de choix, l'ensemble de machines sous-chargées est analysé dans des ordres générés aléatoirement, par chaque composant de correction.

## 13.9 Placement initial en ADAJ

La correction dynamique d'un déséquilibre, pendant l'exécution d'une application, est réalisée en ADAJ par la redistribution des objets déjà instanciés. Une alternative, pour la correction, est le placement initial, lors de l'instanciation d'un objet. Un bon placement initial, généralement, est censé éviter des redistributions fréquentes des objets par la migration. En même temps, le placement de nouveaux objets créés pendant l'exécution de l'application peut corriger les déséquilibres apparus, à cause d'une décision inadéquate de migration ou à cause de l'évolution trop dynamique de l'application.

En ADAJ, le placement initial d'un objet sera décidé en fonction des charges des machines. Les machines les moins chargées seront préférables aux machines normalement chargées, tandis que les machines fortement chargées seront évitées. Dans le cas de l'absence de déséquilibre, donc d'inexis-

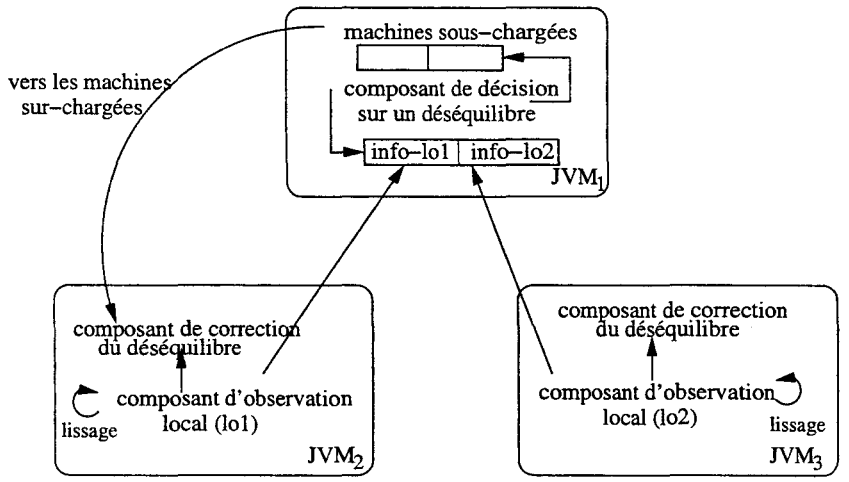



Figure 13.2 – Dispositif général d'équilibrage de charge en ADAJ

tence des machines faiblement et fortement chargées, la politique de placement sera aléatoire.

Le déséquilibre en ADAJ est défini à partir de deux métriques, le nombre de threads et la quantité de travail de la JVM (voir la section 13.5). Cette définition atténue la qualité des métriques, en termes de valeurs. Les machines classifiées sous-chargées ont toutes la même potentialité d'être destinataires des objets. L'intérêt du placement initial est de placer des objets, plus sur les machines sous-chargées, moins sur les machines normalement chargées, et si possible, rien sur les machines sur-chargées, décisions directement liées aux valeurs de charge. Ceci rend nécessaire l'expression de la charge d'une machine virtuelle, comme fonction dépendant de deux métriques, sous forme de fonction d'agrégation.

Deux types de contraintes s'imposent pour la définition de la fonction :

- la relativisation des métriques, pour qu'elles aient un même ordre de grandeur,
- la conformité avec l'identification des groupes des machines (sous, normalement et sur chargées), c'est-à-dire que la valeur de charge d'une machine normalement chargée devrait être supérieure à la valeur de charge d'une machine sous-chargée et inférieure à la valeur de charge d'une machine sur-chargée.

Ainsi, plus la valeur de la fonction de charge est grande, plus la charge de la machine est importante.

### 13.9.1 La fonction de charge de la JVM

Etant donné que la charge dépend du nombre de threads et de la quantité du travail (voir la section 13.4), la fonction en dépendra aussi. La plus simple fonction d'agrégation, qui considère les deux métriques, est la somme pondérée.

Une première contrainte imposée, pour l'utilisation d'une telle fonction d'agrégation, est d'estimer des valeurs sur une même échelle. Pour respecter cette contrainte, les informations de nombre de threads, et de quantités de travail seront relativisées par rapport à leur somme respective.

La classification des machines restent la même, en considérant des pourcentages, au lieu de valeurs brutes. Le diagramme de la figure 13.1 devient le diagramme présenté dans la figure 13.3.

Dans la figure 13.3, les seuils qui déterminent l'intervalle de normalité sont :

- pour le pourcentage de nombre de threads, le seuil inférieur, $UN_{thr}$, le seuil supérieur, $NO_{thr}$,

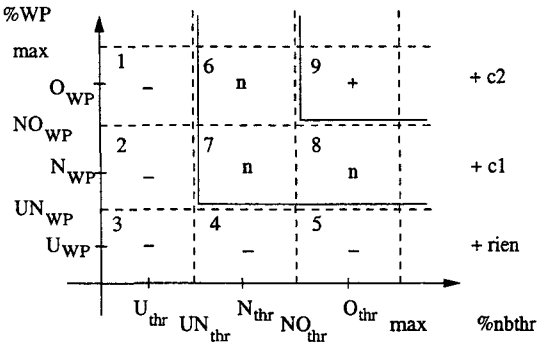



Figure 13.3 – Classification des JVM en fonction des métriques de charge relativisées

- pour le pourcentage de la quantité de travail, le seuil inférieur, $UN_{WP}$, le seuil supérieur, $NO_{WP}$.

Ces seuils sont calculés différemment en fonction des algorithmes décrits dans la section 13.6. Ils dépendent de l'écart moyen ou type pour l'algorithme autour de la moyenne, ou des centres de groupes, pour l'algorithme K-Moyennes. La fonction reste générique, indépendamment de l'algorithme choisi.

La fonction de charge peut être définie, pour la machine virtuelle $JVM_i$, comme $charge(i) = \%WP + \%nbthr + c$, où $c$ est une constante. L'analyse explicitée ci-dessous montrera la valeur de la constante $c$.

Une deuxième contrainte, sur la cohérence des valeurs de la fonction de charge avec le classement des machines, est reflétée dans deux conditions :

$$max\ \ charge(i)_{1-5} < min\ \ charge(i)_{6-8} \tag{13.1}$$

et

$$min\ \ charge(i)_{9} > max\ \ charge(i)_{6-8} \tag{13.2}$$

qui traduisent que la valeur de la fonction de charge pour les machines classées sous-chargées (correspondant aux carrés 1-5 dans la figure 13.3) est inférieure à la valeur de la fonction de charge pour les machines classées normalement chargées (correspondant aux carrés 6-8), et la valeur de la fonction de charge pour les machines classées sur-chargées (correspondant au carré 9) est supérieure à la valeur de la fonction de charge pour les machines classées normalement chargées.

La relation 13.1 est équivalente à

$$max(max_{WP} + UN_{thr}, UN_{WP} + max_{thr}) < UN_{WP} + UN_{thr} + c_1,$$

d'où

$$c_1 > max(max_{WP} - UN_{WP}, max_{thr} - UN_{thr}),$$

où $max_{WP}$ est le pourcentage maximal correspondant à la valeur maximale de la quantité de travail et $max_{thr}$ est le pourcentage maximal correspondant à la valeur maximale du nombre de threads.

La valeur de la constante $c$ peut être, dans ce cas,

$$c = max(max_{WP} - UN_{WP}, max_{thr} - UN_{thr}) + 0,1^4.$$

[4] une des valeurs minimales qui vérifie la relation précédente

La relation 13.2 est équivalente à

$$NO_{WP} + NO_{thr} + c_2 > max(max_{WP} + NO_{thr}, NO_{WP} + max_{thr}),$$

d'où

$$c_2 > max(max_{WP} - NO_{WP}, max_{thr} - NO_{thr}).$$

La valeur de la constante $c$ peut être, dans ce cas,

$$c = max(max_{WP} - NO_{WP}, max_{thr} - NO_{thr}) + 0,1.$$

Donc, la fonction de charge, pour une machine virtuelle $JVM_i$, est définie de la manière suivante :

$$charge(i) = \begin{cases} \%nbthr_i + \%WP_i & \text{si } (\%nbthr_i, \%WP_i) \in 1-5 \\ \%nbthr_i + \%WP_i + c_1 & \text{si } (\%nbthr_i, \%WP_i) \in 6-8 \\ \%nbthr_i + \%WP_i + c_2 & \text{si } (\%nbthr_i, \%WP_i) \in 9 \end{cases}$$

où

$$c_1 = max(max_{WP_i} - UN_{WP_i}, max_{thr_i} - UN_{thr_i}) + 0,1$$

et

$$c_2 = max(max_{WP_i} - NO_{WP_i}, max_{thr_i} - NO_{thr_i}) + 0,1.$$

### 13.9.2 La politique de placement

La politique de placement décide de la machine d'accueil d'un nouvel objet créé, en utilisant la fonction de charge. A priori, plus la machine est chargée, moins elle devrait recevoir d'objets et plus la machine est libre, plus elle devrait admettre des créations locales d'objets. Pour donner la chance à chaque machine d'être choisie comme destinataire, mais en tenant compte, en même temps, de cette corrélation, nous utilisons le principe de la roulette "pondérée", qui est une des méthodes de sélection dans les algorithmes génétiques [Gol94].

#### Principe de la roulette pondérée

Le principe de la roulette pondérée, appelé aussi échantillonage stochastique avec remplacement, se base sur l'idée de donner à chaque individu une probabilité d'être sélectionné, proportionnelle à sa qualité (représentée par une fonction d'évaluation).

Dans la méthode de sélection des algorithmes génétiques, on reproduit sur un disque de périmètre l'unité, un arc de cercle proportionnel à la qualité de l'individu, puis on fait tourner la roulette aléatoirement (i.e. on tire au hasard un nombre entre 0 et 1) et on sélectionne l'individu pour lequel la somme des probabilités de sélection ne dépasse pas le nombre aléatoire généré.

L'algorithme général de la roulette est le suivant :

- (somme) calculer la somme des qualités de tous les individus ($t$),
- (sélection) générer un nombre aléatoire $r$ entre 0 et $t$,
- (boucle) balayer les individus et sommer les qualités des individus au fur et à mesure. Quand cette somme dépasse la valeur aléatoire $r$, l'algorithme est arrêté et l'individu courant est choisi.

Pour l'ensemble de 6 nombres générés aléatoirement, (0,81 ; 0,32 ; 0,96 ; 0,01 ; 0,65 ; 0,42), le processus de sélection entre les individus du tableau 13.1 est montré dans la figure 13.4.

| Numéro ind | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| qualité | 2,0 | 1,8 | 1,6 | 1,4 | 1,2 | 1,0 | 0,8 | 0,6 | 0,4 | 0,2 | 0,0 |
| probabilité de sélection | 0,18 | 0,16 | 0,15 | 0,13 | 0,11 | 0,09 | 0,07 | 0,06 | 0,03 | 0,02 | 0 |

Tableau 13.1 – Exemple d'individus et de leur probabilité de sélection

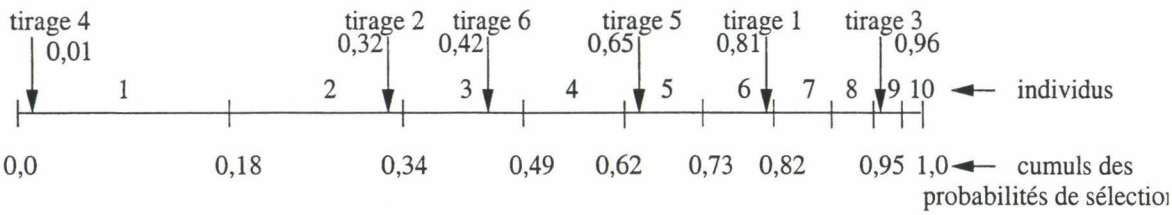



Figure 13.4 – Sélection des individus à base du principe de la roulette pondérée

**Utilité de l'algorithme de la roulette pondérée**

L'algorithme de la roulette pondérée rend une chance de choix, pour un individu, d'autant plus importante que sa qualité est grande. Dans le problème de placement initial des objets, la machine virtuelle, en termes d'algorithme génétique, est un individu. L'intérêt de la roulette serait de choisir une machine dont la charge est dans les moins importantes. Ainsi, la qualité "attachée" à une machine est le maximum entre les valeurs de charge moins sa propre valeur de charge (donnée par la fonction *charge*) :

$$qual(i) = max_j(charge(j)) - charge(i).$$

## 13.10 Conclusions

Le mécanisme d'équilibrage de charge présenté vise le cas des exécutions des applications dans un système dédié, homogène. La définition de charge dérive du nombre de threads et d'une nouvelle mesure, la quantité de travail d'une machine virtuelle Java. Celle-ci permet de différencier les threads bloqués (*non activables*) et les threads prêts à s'exécuter (*activables*). Deux techniques sont employées pour acquérir une bonne distribution de la charge : la redistribution et le placement d'objets.

Le mécanisme de redistribution d'objets est centralisé et périodique pour la collecte des informations locales de charge, et distribué pour les politiques de sélection et localisation. Le déclencheur d'une redistribution est dépendant de la politique de décision, donc plusieurs algorithmes ont été proposés. Ces algorithmes utilisent des informations statistiques sur l'éparpillement des valeurs de charge, à base de la moyenne et de l'écart (moyen ou type). La politique de décision est inspirée de la technique à double seuil, étant dépendante de la moyenne des valeurs analysées.

Le placement d'objets complète le mécanisme de redistribution d'objets pour éviter des mauvais placements aléatoires, ou pour aider à corriger les déséquilibres de charge. Cela est réalisé grâce à une fonction d'évaluation de la charge d'une JVM. Cette fonction est utilisée pour définir la qualité de sélection d'une machine dans un algorithme de roulette pondérée.

Le chapitre suivant présente la mise en œuvre de ces mécanismes dans l'architecture de l'exécutif.

# Chapitre 14

# L'architecture de l'exécutif

Ce chapitre décrit les différents aspects d'implémentation liés à l'environnement d'exécution ADAJ. Si l'environnement de programmation offre une bibliothèque de développement d'applications parallèles et distribuées, l'environnement d'exécution permet l'intégration d'un mécanisme d'équilibrage de charge, compatible avec Java. Les différents composants qui forment ce mécanisme sont présentés, ainsi que leur interactions et leur prise en compte lors du lancement d'une application.

Une fonctionnalité importante est mise-en-avant, la migration des objets, une des méthodes utilisées en ADAJ (en association avec le placement des objets) pour distribuer la charge. La technique JavaParty pour réaliser cette migration impose certaines contraintes, la plus importante étant liée à l'état d'un objet migrable. Seul l'objet passif, qui n'a pas de méthode en exécution, peut migrer. Le mécanisme ADAJ de migration propose une extension de cette technique, par migration d'objets actifs, avec une méthode en cours d'exécution. Dans ce cadre, les fragments interviennent, aussi bien comme des entités de calcul parallèle, que des entités de placement et de migration, leur comportement étant maîtrisé par la bibliothèque proposée. Le surcoût des deux techniques, JavaParty et ADAJ, est mesuré également.

## 14.1 L'environnement d'exécution ADAJ

L'environnement d'exécution ADAJ s'appuie sur celui de JavaParty, constitué d'un composant central, le *Runtime Manager* (une machine virtuelle) et d'une à plusieurs machines virtuelles de calcul.

Lors du lancement de l'application, une nouvelle machine virtuelle Java est créée qui accueille le programme principal. Cet environnement est enrichi pour rendre possibles les différentes politiques du mécanisme d'équilibrage de charge :

**La JVM du programme principal** contient un composant de décision, qui est responsable de
- la collecte des informations de charge,
- la détection d'un déséquilibre,
- la diffusion des informations nécessaires aux machines sur-chargées.

**Chaque machine de calcul** comporte un composant d'extraction de la charge et un composant de correction.

- le composant d'observation comprend un composant d'observation des relations (voir la section 12.2) et un composant d'observation de la charge, qui complète les informations sur la charge de la JVM.
- le composant de correction a une activité de redistribution d'objets pour les machines surchargées, communiquant avec le composant d'observation local et les composants d'observation des relations d'autres machines.

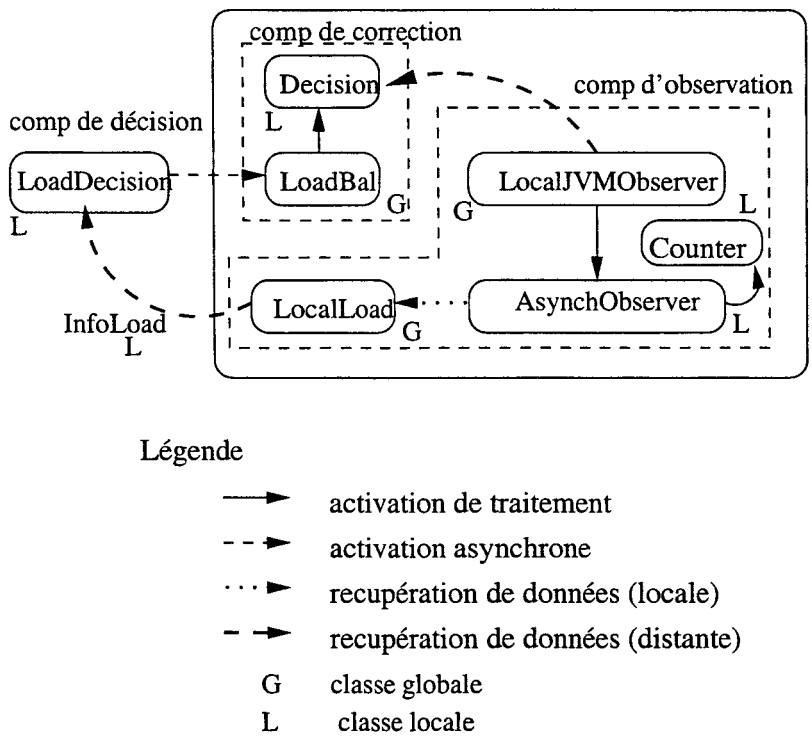



Figure 14.1 – Composants du mécanisme d'équilibrage de charge ADAJ

### 14.1.1 Le composant d'observation

Les objets observés d'une machine virtuelle sont contenus dans une liste gérée par l'observateur de relations local à la machine. Cette liste est réactualisée lors

- de la création d'un objet global sur la machine, ou
- de sa disparition, par le mécanisme de migration, ou
- de l'arrivée d'un objet, suite à une migration.

L'accès à cette liste est réalisé grâce à la classe ADAJ *LocalJVMObserver*.

```
class LocalJVMObserver {
   Hashtable getObservedObjects();
   // renvoie une table avec les objets observés
   // de la machine virtuelle courante
}
```

La table de hachage retournée par la méthode *getObservedObjects* contient une liste d'associations (clé, valeur), où la clé est une référence vers un objet global observé de la machine virtuelle courante et la valeur correspondante contient les valeurs des compteurs associés à l'objet.

Les compteurs correspondant aux relations sont regroupés dans une classe ADAJ, *Counter*. La classe contient, parmi d'autres, des méthodes d'accès aux valeurs des compteurs. La signature de ces méthodes est présentée ci-dessous.

```
class Counter {
   public float getOGIScan();
   // renvoie la valeur lissée du compteur OGI correspondant à
   // à la relation OGI entre l'objet courant et l'objet passé en paramètre

   public float getOLIScan();
   // renvoie la valeur lissée du compteur OLI pour l'objet courant

   public float getIIScan();
   // renvoie la valeur lissée du compteur II pour l'objet courant
}
```

La communication entre le composant de décision et les composants d'observation est réalisée grâce à un objet distant, instance de la classe *LocalLoad* se trouvant sur chaque JVM enregistrée dans l'environnement. Cette classe fournit des informations sur le nombre de threads ou sur la quantité de travail local de la JVM, basée sur les dernières mesures lissées des relations entre les objets globaux. Les informations sont regroupées dans une classe locale, *InfoLoad* :

```
public class InfoLoad implements Serializable {
    public int nbThreads;
    public float sumWP;
}
```

### 14.1.2 Le composant de décision

Les informations de charge issues des composants d'observation sont regroupés dans le composant de décision, instance de la classe *LoadDecision*. Cet objet active de manière asynchrone, les composants de correction correspondant aux machines sur-chargées, détectées par la politique de décision qu'il applique.

### 14.1.3 Le composant de correction

Les machines sur-chargées, détectées par le composant de décision, sont informées de leur état par le biais d'un objet de la classe *LoadBal*, classe globale, non-observée. Celui-ci traite la situation de surcharge, par l'intermédiaire de la classe *Decision*, classe locale, ayant uniquement des méthodes statiques. Vue sa fonctionnalité, de mettre en œuvre la politique de sélection et de transfert, la classe a des références également sur les composants distants d'observation de relations.

## 14.2 L'exécution d'une application ADAJ

### 14.2.1 Génération de code binaire (bytecode)

Une classe globale est soumise à deux types de transformations : la pré-compilation, par le compilateur JavaParty et la post-compilation, par un compilateur ADAJ.

La pré-compilation, concernant le code source de la classe à transformer, génère des classes qui rendent les fonctionnalités d'une classe remote, dont l'accès à distance des objets remote, la

migration et la gestion de la partie statique. Les classes générées, au nombre de 10, sont montrées dans la figure 14.2, pour la classe remote initiale $A$.

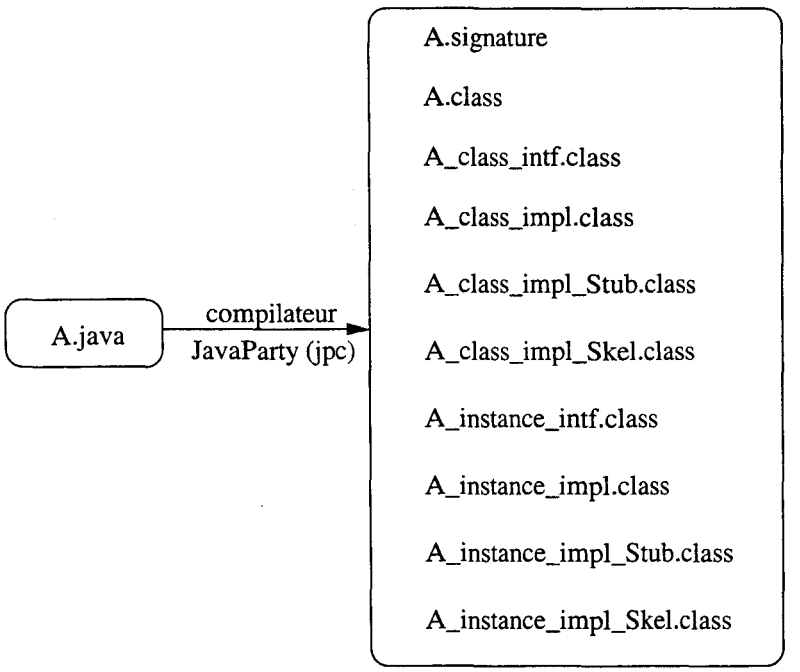



Figure 14.2 – Classes générées par le compilateur JavaParty

La post-compilation concerne uniquement le bytecode de la classe (*A.class*) et les bytecodes des classes gérant la partie instance (*A_instance_intf.class*, *A_instance_impl.class* dans la figure 14.2). La transformation implique l'ajout des instructions de type bytecode pour compter le nombre d'invocations.

Les bytecodes des classes utilisées pour l'exécution d'une application ADAJ sont disponibles dans toutes les machines virtuelles Java, par l'utilisation d'un même système de fichiers.

### 14.2.2 Lancement d'une application ADAJ

Pour prendre en compte les différents paramètres des stratégies de distribution, des arguments supplémentaires sont nécessaires pour l'exécution d'une application ADAJ :

– la période de lissage des informations de relations ($t$) $\rightarrow$ $-relation : sleepTime = t$,

– le déclenchement ou pas du mécanisme de distribution de charge,
– la stratégie appliquée pour la distribution de charge (*algo*),
– le ou les seuil(s) pour les algorithmes à seuil (*seuil*),
– la fréquence de la politique d'information de charge ($\Delta t$).

$\rightarrow$ $-lb : algo : seuil : inspTime = \Delta t$

## 14.3 La mise en œuvre de la migration

Cette section présente les aspects liés à la migration de processus, la liaison avec la migration d'objets et les techniques de la réalisation de celle-ci en JavaParty et en ADAJ. L'évaluation de leur coût ou surcoût est présentée.

### 14.3.1 Migration de processus

#### Motivation de la migration

Différentes recherches ont été menées dans le domaine de mobilité, en termes de migration de processus dans un système distribué, les techniques de mobilité ayant comme objectif la reconfiguration dynamique des applications, le nomadisme de l'utilisateur, l'administration du système ou la répartition de charge.

**La reconfiguration d'une application** [KGA+00, PBR99] répond à des modifications à faire dans l'architecture de l'application, suite à des suppressions ou apparitions de composants logiciels.

**Le nomadisme** [BCKP95] caractérise les systèmes à agents mobiles, où la communication permet de suivre les migrations. Pourtant, la plupart des systèmes Java mobiles existants n'offrent pas de migration de threads. Les agents mobiles, qui permettent la migration d'agents, introduisent un surcoût important dans l'exécution.

**L'administration des systèmes distribués** exige parfois la ré-initialisation des machines, avec la contrainte d'assurer la continuité de certains services. Le transfert de ces services sur d'autres machines s'impose et se réalise grâce au mécanisme de migration.

**La répartition de charge,** [LLM88] pendant l'exécution d'une application distribuée, utilise la migration, pour mieux exploiter les ressources, de manière uniforme. Des applications, s'exécutant dans un système distribué, nécessitent une gestion judicieuse des ressources, afin de profiter de nouvelles ressources physiques (mémoire ou processeur), de disponibilité des ressources existantes ou, encore, d'équilibrer la charge entre les machines du système distribué. La mobilité de l'application permettra d'améliorer l'exécution de l'application.

#### Algorithme général de migration de processus

Les différentes techniques de réalisation du mécanisme de migration de processus considèrent les étapes suivantes :

- le processus est suspendu, dans son exécution, sur la machine source,
- le contexte d'exécution du processus est récupéré,
- un nouveau processus est créé sur la machine destinataire, qui s'exécutera dans le contexte d'exécution extrait,
- le contexte d'exécution extrait est envoyé au nouveau processus créé,
- le nouveau processus continue son exécution, dans le contexte qu'il a reçu.

De plus, la migration de processus doit prévoir des mécanismes de contrôle des communications. Ainsi, les requêtes vers le processus migré devraient être dirigées vers la nouvelle localisation du processus, pour être traitées après la migration.

#### Problèmes rencontrés lors de la migration de processus

Les principaux problèmes soulevés par la migration de processus concernent le transfert de l'état et/ou du contexte d'exécution du processus et le maintien des communications entre processus. Nous détaillons ci-dessous chacun de ces problèmes.

**Le contrôle de la migration.** La migration de processus doit prendre en compte différents aspects dont l'unité de migration et le mécanisme de transfert de l'état du processus. L'unité de migration, qui définit la *granularité*, est, en général, le processus lourd [TLC85, OTCH94, PM83]. Le système V est basé sur l'échange de messages entre les stations de travail. En V [TLC85], un "hôte logique" migre ; il est l'équivalent d'un espace d'adresses, avec le processus et ses processus fils. DEMOS/MP [PM83], basé aussi sur la communication des messages, migre la mémoire du processus contenant le code, les données et la pile. Accent [Zay87] utilise le mécanisme de mémoire virtuelle (*copie sur référence*) pour effectuer la migration. Le transfert du contexte s'exécute au départ, et le transfert des données est réellement effectué lorsque le destinataire utilise explicitement les données. Pour toute migration de processus, la migration du code est implicitement réalisée. Condor [LLM88] propose une technique différente de migration, en utilisant un fichier de sauvegarde.

En général, il n'existe pas de contraintes particulières pour pouvoir migrer des processus. Pourtant, des systèmes imposent des contraintes pour que la migration soit réussie. En Chorus [OTCH94], par exemple, un processus est suspendu, pour la migration, dans un état bien défini, n'ayant pas de processus fils, suspendus, qui exécutent des appels systèmes.

**L'incidence de la migration sur les communications.** Dans un système distribué, les processus communiquent pour effectuer du travail. Le maintien des communications est exigé lors de la migration d'un des processus communiquant. Ainsi, des politiques de localisation du processus migré doivent être prévues. Ces techniques de localisation montrent comment la migration affecte les processus avec lesquels le processus migré communique et comment sont gérés les messages arrivant pendant la migration. Les différentes politiques, après MOA [MLC98], dans le cadre des systèmes à agents, sont les suivantes :

- la sauvegarde de la trace de la localisation d'un processus sur une machine hôte, et sa mise à jour, lors de la migration du processus,
- l'enregistrement dans un serveur de noms,
- la recherche à partir d'un itinéraire,
- le mécanisme de forwarding (Des traces sont laissées lors de la migration).

Une des techniques les plus fréquentes, le forwarding (ou redirection des messages), consiste à sauvegarder sur la machine source la nouvelle localisation du processus migré (donc la machine destinataire). Au départ, les processus ayant des références sur le processus migré ne sont pas informés de sa migration ; ainsi, les messages qu'ils continuent à envoyer vers la localisation précédente seront redirigés vers la nouvelle localisation du processus.

Ce mécanisme présente l'inconvénient d'allonger considérablement la chaîne de *dépendance résiduelle*. Le degré de dépendance résiduelle caractérise la longueur de la liste d'adresses d'un processus correspondantes à ses localisations. Lors de son arrivée sur une machine déjà visitée, les communications vers le processus migré devraient arrivées directement, et pas être propagées par la chaîne d'adresses. Deux techniques existent pour réduire les communications dans ce cas :

- la redirection sur une machine déjà visitée est effacée,
- la mise à jour de la chaîne avec la nouvelle localisation du processus toute de suite après l'identification de la localisation du processus.

Le système Accent [Zay87], à cause du contrôle particulier de la migration, ne propose pas de technique de localisation, DEMOS/MP [PM83] choisit la technique du forwarding pour acheminer les messages arrivés pendant la migration. Dans Charlotte [AF89], MOSIX [BLS99] ou Mach [MZD92], les messages ne sont pas délivrés, mais gardés chez la source, jusqu'à ce que le processus migré ait établi sa destination. En Chorus [OTCH94] ou Amoeba [MvRT+90], la recherche de la localisation du processus migré est réalisée par une diffusion.

### 14.3.2 Migration d'objets

La migration d'un objet d'une JVM à une autre, impose de réaliser un transfert, sur une autre machine, qui permet que toute nouvelle invocation sur l'objet soit exécutée sur le nouvel placement. De plus, tout appel arrivant pendant ou après la migration devrait être correctement acheminé vers la nouvelle localisation. Une description formelle à l'aide des chaînes de Markov de l'acheminement des communications lors de la migration d'un objet se trouve dans [Hue02]. La migration d'objets s'inspire du modèle de programmation à processus, où différentes stratégies de migrations sont offertes.

La migration d'objets revient à la migration de processus, si l'objet à migrer a, au moins une méthode en cours d'exécution (celle-ci étant exécutée dans un thread). Dans ce cas, migrer un objet, fonctionnellement, coïncide avec la migration du processus léger (thread) dans lequel la méthode s'exécute.

#### Algorithme de la migration d'objets

La migration d'un objet consiste à le transférer de la machine l'hébergeant (*machine source*) vers une autre machine (*machine destinataire*), en conservant la cohérence de son état. Si une méthode est en exécution sur l'objet, les étapes suivantes sont parcourues :

- l'interruption de l'exécution de la méthode,
- le transfert, de la machine source vers la machine destinataire, de l'état du programme (le bytecode), de l'état des données (la valeur courante des données), et de l'état d'exécution,
- la reprise de l'exécution de la méthode sur la machine destinataire, à partir des données transmises.

#### Techniques de migration

Les différentes techniques de migration sont dues principalement au transfert du contexte d'exécution considéré lors de la migration. Les critères pris en compte pour la classification des techniques de migration sont : le degré de la mobilité, l'hétérogénéité ou la portabilité.

- degrés de mobilité : exécution à distance ou code à la demande ; migration faible ou migration forte,
- hétérogénéité : représentation de l'état d'exécution d'un processus, indépendante ou pas de la machine,
- portabilité : mécanisme intégrant un système existant ou, dans un nouveau système, offrant, entre autres fonctionnalités, la migration d'objets.

Certaines caractéristiques de la migration sont à considérer et à analyser : les types de migration, le mode d'initiation de la migration, la granularité, l'hétérogénéité et les types de la mise en œuvre.

**Les types de migration.** Les concepteurs des mécanismes de mobilité décrivent deux types différents de migration : *la migration faible* et *la migration forte.*

**La migration faible** suppose le redémarrage de l'exécution interrompue, sur la machine destinataire, depuis le début. Ce type de migration est aussi appelée *non-transparente.* La cohérence de l'état de l'objet n'est assurée que s'il n'existe pas de méthode s'exécutant sur l'objet. Dans le cas contraire, reprendre la méthode depuis le début risque d'exécuter deux fois des instructions. Plusieurs systèmes [PZ97], [IBM] fournissent de la mobilité faible, s'appuyant sur le mécanisme de sérialisation Java [Sung], soit en vue de l'optimisation de l'exécution, soit pour rendre mobiles les

agents. Le mécanisme proposé par JavaParty, présenté dans la section 3.3.3 de cette thèse, propose de la migration faible et sera évalué dans la section 14.3.3.

**La migration forte** prend en compte l'état courant de l'exécution de la méthode interrompue. Ainsi, une méthode ne reprendra pas son exécution sur la machine destinataire depuis le début, mais poursuit au point même où elle a été interrompue. Des plates-formes à agents mobiles, comme Sumatra [RADS97] ou MAP [Per97], permettent la mobilité forte pour leurs agents. Les travaux prenant en compte l'état courant de l'exécution des applications sont moins nombreuses, à cause de la complexité de la mise en œuvre.

**Les modes d'initiation.** Une caractéristique importante de la mobilité est son mode d'initiation. Les différentes approches s'appuient sur le type de déclenchement de la migration, interne ou externe. Ainsi,

- si l'objet initie sa propre mobilité, le mode d'initiation est appelé *forcé*,
- si une entité externe initie la mobilité de l'objet, le mode est appelé *décidé*.

*La mobilité forcée* se traduit par un appel, sur l'objet, d'une primitive de migration par une entité externe et son application à l'objet mobile. Ce type de mobilité trouve son utilité dans l'automatisation de la répartition dynamique de charge ou à la reconfiguration dynamique d'une application [WTV02, MWL99].

*La mobilité décidée* se traduit par un appel à une primitive de migration de l'objet lui-même. Cependant il faut noter que cette mobilité est applicable dans les systèmes à agents [SMY99, Bou99, Fün98], où les agents sont des entités autonomes qui prennent l'initiative de leurs propres déplacements.

Ici, en ADAJ, c'est une entité extérieure que invitera l'objet à faire un appel à sa propre primitive.

**La granularité.** La *granularité* qualifie la mobilité en fonction du nombre d'objets qui migrent à la fois. Avec une granularité à gros grain, la migration d'un objet entraîne la migration d'autres objets résidant sur la même machine. Par exemple, le système Merpati [Sue00] propose un service de mobilité forte en Java, où la mobilité suppose le déplacement de l'ensemble des threads Java résidant sur la JVM, vers la machine destinataire. A l'opposé, une mobilité à grain fin de parallélisme repose sur le déplacement d'un objet indépendamment des autres objets résidant sur la machine. Par exemple, JavaParty propose une mobilité faible, à granularité fine pour les objets de ses applications.

**L'hétérogénéité** La mobilité d'une application a été analysée sur deux types différents de réseaux : homogène et hétérogène.

Dans un réseau homogène, le transfert de l'état d'un objet ne soulève pas de problèmes particuliers, car une même architecture est employée et un même système d'exploitation est utilisé.

Dans un réseau hétérogène, le problème de représentation de l'état transféré apparaît, en fonction de sa dépendance du système sous-jacent. Si la représentation de l'état transféré dépend du système sous-jacent, variant d'une plate-forme à l'autre, l'état ne peut pas être transféré directement. Des transformations supplémentaires s'imposent.

Un premier type de transformation consiste à traduire la représentation de l'état, d'un format de la plate-forme source à un format de la plate-forme destinataire [Shu90]. Cette solution doit prendre en compte toutes les possibilités de combinaisons possibles des deux types de plates-formes.

Une deuxième solution, moins coûteuse, est de traduire les formats sur les plates-formes sources et destinataires, dans des formats intermédiaires, standard. L'avantage de cette approche est la

construction des traducteurs seulement pour les nouvelles plates-formes utilisées. Cette solution a été adoptée par le système Tui [SH98], pour la migration de processus entre des plates-formes hétérogènes.

Quant aux représentations indépendantes de la plate-forme sous-jacente, gérer l'hétérogénéité lors de la migration revient à développer un mécanisme de migration sur un système homogène de machines, toutes créés sur le modèle d'une machine virtuelle. C'est l'approche proposée en Java, par l'utilisation de la machine virtuelle Java.

**Les types de mise en œuvre.** La migration d'un thread consiste à pouvoir transférer trois types d'information : le code, les données et l'état de l'exécution. Java permet :

- la migration de code, par le téléchargement dynamique du bytecode par la JVM [SUNa],
- la migration de données, par le mécanisme de sérialisation.

Par contre, la migration de threads n'est pas totalement possible dans la technologie Java, à cause de l'impossibilité de migration de l'état d'exécution. L'information manquante se trouve dans la machine virtuelle, que la sécurité Java interdit de récupérer.

Trois solutions ont été proposées pour prendre en compte la migration de threads Java :

- une extension de la machine virtuelle Java (en la modifiant),
- une extension du langage Java (en introduisant un pré-processeur),
- une extension du compilateur (par transformation du bytecode).

**L'extension de la JVM** suppose que les applications faisant appel à la migration s'exécutent sur des JVM modifiées. L'inconvénient majeur de cette approche est la non-portabilité du support implémenté dans la plate-forme Java existante. Ce type de mise en œuvre est proposé par S. Bouchenak dans [BH00, Bou01].

**L'extension du langage** est une technique qui ne permet pas d'extraire entièrement l'état de l'exécution d'un thread (comme les valeurs sur la pile d'opérandes). L'efficacité d'une telle méthode est limitée par l'utilisation des constructions de type `if imbriqué` pour simuler des instructions de type `goto` qui permettent d'éviter la reprise du code déjà exécuté. L'inefficacité et l'inflexibilité de l'approche la rend limitée, en comparaison avec d'autres. De plus, le code source n'est pas toujours disponible pour appliquer cette technique. Pourtant, elle est utilisée dans les projets Wasp [Fün98] et JavaGO [SMY99].

**L'extension du compilateur,** au niveau de la post-compilation, se montre plus efficace que l'extension du langage. Un des avantages de l'utilisation de la transformation du bytecode réside dans la possibilité d'insertion des instructions de transfert de contrôle (du genre `goto`) dans le bytecode. Le langage Java ne permet pas ce type de construction et pour l'introduire, la taille du bytecode inséré augmenterait considérablement. Les difficultés d'une transformation de bytecode se trouvent dans la vérification du bytecode modifié et dans l'identification des valeurs à sauvegarder et à restaurer. Les travaux sur la modification du bytecode sont menés dans les projets Brake [TRC+00], JavaGOX [SSY00] ou Moba [vLSM00].

### 14.3.3 Migration JavaParty

#### Considérations générales

La migration d'un objet remote en JavaParty (voir aussi la section 3.3.3) ne peut être effectuée que si, et seulement si, aucune méthode ne s'exécute sur l'objet remote, et si l'objet n'a pas été

marqué non-migrable. Pour tester la première contrainte, un compteur dénombre chaque appel de méthode sur la partie instance (la partie statique, donc l'objet classe, a une localisation fixe et donc, n'intervient pas dans le mécanisme de migration). Au début de la méthode, le compteur est incrémenté et à la fin de la méthode, caractérisée par le bloc *finally*, le compteur est decrémenté. Le marquage d'un objet remote comme non-migrable est réalisé au niveau de la déclaration de classe et est indiqué par un attribut statique de la classe.

Les étapes de la migration d'un objet remote sont illustrées dans la figure 14.3.

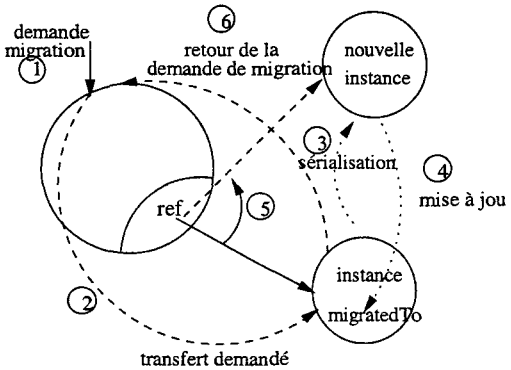



Figure 14.3 – Le schéma de migration d'un objet remote

Une demande de migration d'un objet remote (opération 1) est reportée sur la partie instance (référencée par l'attribut *ref* - opération 2). L'objet instance vérifie que les deux conditions de migrabilité sont satisfaites. Dans le cas affirmatif, la partie instance se sérialise et se transfère sur la machine destinataire (opération 3). La nouvelle référence de l'objet instance est sauvegardée dans l'attribut *migratedTo* (opération 4).

La migration doit également assurer que tout appel arrivant avant, pendant et après la migration s'exécute effectivement. L'acheminement des appels lors d'une migration des objets remote en JavaParty se base sur le mécanisme de forwarding. Ceci est réalisé par la mise à jour de la référence interne du proxy (*ref*) qui pointe vers l'objet instance (opération 5 dans la figure 14.3).

Si un appel arrive pendant le transfert (réalisé par les opérations 3 et 4), l'incrémentation du compteur d'appels est bloquée ; c'est une opération synchronisée sur l'objet instance, tout comme le transfert. Une fois le transfert effectué avec succès, l'attribut *migratedTo* a une valeur non-nulle ce qui provoque une exception lors de l'incrémentation du compteur de méthodes. Ainsi, l'arrivée de l'appel détecte la migration de l'objet, la nouvelle référence est actualisée, l'attribut *migratedTo* remis à nul et l'appel est relancé.

### Evaluation du coût de la migration

La migration d'un objet introduit un coût, que la migration soit réussie ou non. Si la migration échoue, dans le cas d'une méthode en cours d'exécution sur l'objet, le coût est négligeable, parce que la première étape de la migration, la sérialisation, n'est pas encore engendrée. Le coût peut être estimé au temps pris par le test de la valeur d'un attribut, qui décompte le nombre de méthodes en cours d'exécution. Dans le cas d'une migration réussie, deux types de tests ont été effectués pour estimer le temps de migration d'un objet :

- la migration d'un objet vide (sans attributs),
- la migration d'un objet non-vide (avec attributs).

Le premier test évalue le coût d'une migration brute, où la sérialisation est presque inexistante. Le second test estime le coût de la migration, quand la sérialisation n'est pas négligeable. Cette estimation permet d'évaluer le surcoût dû à la sérialisation uniquement.

L'objet non-vide considéré a différentes configurations :

- un objet vector contenant 100 objets nuls,
- un objet vector contenant 100 objets de type `Integer`,
- un objet vector contenant 100 objets, chacun ayant 100 objets de type `Integer`.

Les deux tests consistent dans l'exécution d'un programme Java (version JDK1.3) ayant une boucle ($n$ itérations) qui effectue :

- la création d'un objet, sur la machine virtuelle numéro 0,
- la migration de l'objet, sur la machine virtuelle numéro 1,

sur un environnement formé par deux machines virtuelles. Les temps sont mesurés avant et après la migration. Pour éviter les éventuelles fluctuations qui apparaissent dans un environnement distribué, plusieurs valeurs de $n$ ont été testées : 20, 30, 40 et 50 et plusieurs exécutions (10) du même programme ont été lancées : les temps d'exécution, les plus rapides et les plus lents, ont été écartés de la moyenne calculée.

Deux types d'environnements ont également été testés :

- un environnement homogène, formé par deux machines, à processeur PIII à 733 MHz et 128 Mo de capacité mémoire, sous la plate-forme Linux 2.2.17 (Debian 2.2), liées par un réseau de débit 100 Mb/s,
- un environnement hétérogène, formé par deux machines, une à processeur PIII à 600 MHz, 128 Mo RAM, sous Linux 2.2.17 avec la version jdk1.3.0rc1 Java et l'autre, un processeur PIV à 1,66 GHz, 256 Mo RAM, sous Linux 2.2.2_2, avec la version 1.3.1 de Java, liées par un réseau de débit 10 Mb/s.

Les résultats des évaluations sont présentés dans le tableau 14.1, pour $n = 40$ dans le réseau homogène et pour $n = 50$ dans le réseau hétérogène (les valeurs de $n$ spécifiées correspondent à la stabilité des valeurs et les autres résultats se trouvent dans l'annexe D).

| *temps (ms)* | *réseau homogène* ($n = 40$) | *réseau hétérogène* ($n = 50$) |
|---|---|---|
| objet vide | 13,8 | 21,6 |
| objet à 100 objets nuls | 14,6 | 23 |
| objet à 100 Integer | 16,6 | 26,5 |
| objet à 100 objets chacun à 100 Integer | 280,8 | 277,5 |

Tableau 14.1 – Coûts (en ms) de la migration JavaParty dans un réseau homogène et hétérogène

Les résultats montrent que le coût de migration est essentiellement dû au coût de sérialisation (pour le réseau hétérogène) :

- pour un objet contenant 100 objets nuls - 1,4 ms,
- pour un objet contenant 100 objets de type `Integer` - 5 ms,
- pour un objet contenant 100 objets, chacun à 100 objets de type `Integer` - 250 ms.

Ainsi, le coût de la migration croît avec la taille de l'objet à migrer, mais pas forcément de manière linéaire (le temps de la migration pour un objet de 100 objets, chacun à 100 objets Integer est 10 fois plus grand que le temps de migration d'un objet ayant 100 fois moins de valeurs).

La comparaison directe entre les temps de migration pour les deux types de réseau est biaisée à cause de la puissance différente des machines concernées dans le transfert pour le réseau hétérogène. Pour des objets de taille négligeable, le coût de la sérialisation, respectivement de la désérialisation n'est pas important, et en conséquence c'est la latence réseau qui fait la différence. Si l'objet a une taille plus grande, le coût de la sérialisation ou désérialisation augmente et dépend directement de la puissance de la machine exécutant l'opération, recouvert (pour le réseau hétérogène) ou non (pour le réseau homogène) par la latence réseau.

### 14.3.4 Migration ADAJ

Le problème de la mobilité des applications est abordé ici, en ADAJ, en vue de la répartition dynamique de la charge.

#### Motivation

La migration en JavaParty s'applique à tout objet remote sans aucune méthode en cours d'exécution. En ADAJ, la propriété de migration est essentiellement intéressante pour les fragments, qui constituent la granularité d'un traitement. L'équilibrage de charge vise la répartition des objets possédant la capacité de migration. Mais, une contrainte, imposée pour une migration réussie en JavaParty, empêche la migration d'un objet sur lequel une méthode a commencé à s'exécuter. Plus cette méthode est longue en temps d'exécution, moins l'objet sera capable de migrer.

#### Objectif

L'objectif d'ADAJ est de fournir un mécanisme de migration d'objets ayant une méthode en cours d'exécution. Tenant compte qu'une technique de migration est déjà offerte en JavaParty, il serait intéressant de pouvoir l'exploiter, avec l'objectif de l'étendre. L'intérêt est de ne pas modifier le mécanisme de JavaParty. Mais du coup, tous les objets remote ne pourront migrer avec la méthode ADAJ. Seuls les fragments, dont le comportement est maîtrisé, sont considérés.

Les fragments sont des objets remote multithreadés : en général, au moins un thread est en cours d'exécution sur un fragment. La migration ADAJ des fragments ne coïncide pas avec la migration d'un thread, car un thread peut contenir une exécution de plusieurs méthodes démarrées (méthodes imbriquées). Donc, en ADAJ, la migration ne s'applique que pour les fragments avec une seule méthode et un seul thread en cours d'exécution.

#### Mécanisme de migration ADAJ

Le mécanisme de migration d'objets actifs est basé sur la migration des threads. Pour permettre la migration de threads Java, Bouchenak [Bou98] identifie trois étapes : l'extraction du contexte d'exécution du thread, le transfert de ce contexte de la machine source vers la machine destinataire et l'intégration de ce contexte à un nouveau thread d'exécution dans la machine destinataire.

En ADAJ, chacune des trois étapes de la migration se retrouve, sans que la JVM soit modifiée, mais en modifiant le bytecode, par une post-compilation. Les étapes sont présentées par la suite.

**Extraction du contexte d'exécution du thread.** Le contexte d'exécution d'un thread Java sera constitué d'une image de son exécution, contenant des informations nécessaires à la restitution du contexte. Un thread Java est caractérisé par une pile, un tas d'objets et la zone de méthodes.

La sauvegarde de ces informations est censée construire des structures de données indépendantes de la machine. Grâce à la propriété de masquer l'hétérogénéité d'un système, les classes Java sont une solution.

**La pile Java,** l'équivalent de la pile d'exécution utilisée dans un langage conventionnel tel que le langage C, permet la sauvegarde des variables locales et des résultats intermédiaires ; elle sert aussi à empiler les appels de méthodes imbriquées. Cette pile donne accès aux objets et classes Java manipulés par le thread. La sauvegarde de la pile suppose la construction d'une liste de noms et de références de classes Java et d'une liste de références vers les objets utilisés, retrouvées dans les tables de variables et dans les piles d'opérandes contenues dans les frames[1]. Le problème apparu concerne les types de ces variables. Les types ne peuvent pas être connus de manière statique. La solution réside dans le typage des instructions bytecode qui s'applique à un type défini de valeurs. Ainsi, une pile supplémentaire, avec les types de chaque opérande, est construite, au moment de la sauvegarde de l'état du thread.

**Le tas d'objets** rassemble les objets manipulés par un thread. La pile du thread gère les références vers les objets se trouvant dans le tas. La récupération de ces objets est implicitement réalisée par le mécanisme de sérialisation Java, qui parcourt le graphe d'objets obtenu par les références des objets et les stocke dans une structure qui permet la reconstitution du graphe d'objets.

**La zone de méthode** contient, pour chaque classe, le code de ses méthodes et les constructeurs. Tenant compte que le code est supposé exister sur toutes les machines et, éventuellement, pouvoir être chargé dans la machine virtuelle Java, aucun traitement auxiliaire n'est imposé pour la zone de méthode.

**Le transfert du contexte d'exécution.** L'objectif est ensuite de transférer le contexte d'exécution de la machine source vers la machine désignée destinataire. Ce transfert est réalisé en transportant la structure de données construite lors de l'extraction du contexte d'exécution.

**L'intégration du contexte d'exécution à un nouveau thread.** L'intégration du contexte transféré consiste à créer, sur la machine destinataire, un nouveau thread, dont le contexte est initialisé avec le contexte transféré et lancer son exécution, par une reprise du point où il a été interrompu. Les opérations sont similaires à celles réalisées pour la sauvegarde de l'état d'exécution : la pile d'exécution est rétablie, en reconstituant les valeurs, en fonction de leur type, le tas d'objets est recréé, par la désérialisation. Le mécanisme doit prévoir la reprise de la méthode à exécuter depuis son point d'interruption, par l'interprétation par la JVM des instructions de bytecode constituant la suite du code qui reste à exécuter.

La migration JavaParty réalise le transfert du contexte d'exécution dans le cas où la pile Java est vide, donc il n'existe pas de thread en exécution. Le mécanisme de migration ADAJ (figure 14.4) ajoute des traitements particuliers pour la sauvegarde et l'intégration du contexte d'exécution (notamment la pile), le transfert du contexte étant réalisé de manière implicite, grâce au mécanisme de migration JavaParty.

[1] un frame est une zone de mémoire attachée à chaque appel de méthode Java

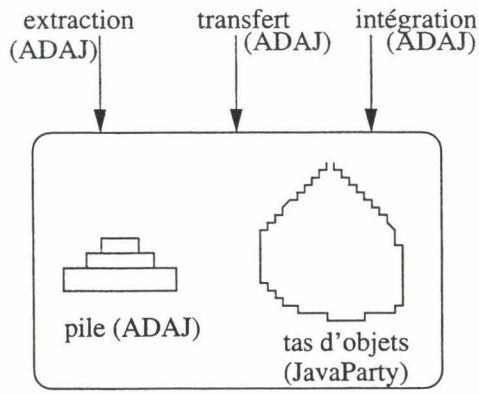



Figure 14.4 – Migration ADAJ

### Principes de la mise en œuvre de la migration ADAJ

La migration en ADAJ est réalisée par une transformation de bytecode, grâce à la facilité d'exprimer des instructions de type `goto` dans le bytecode, par rapport au code source. La migration en JavaParty est forcée et en ADAJ, grâce à l'utilisation du mécanisme d'équilibrage de charge, elle reste forcée, donc décidée à l'extérieur de l'objet migré. Cette décision est signalée à l'objet. Dès que l'objet détecte la demande de migration qui lui a été faite, il engendre sa migration. La détection de ce changement d'état ne peut pas être réalisée à toute instruction du code source, ou du bytecode. Elle sera faite à certains endroits, appelés *points de migration.*

L'approche en ADAJ choisit les débuts de boucles comme points de migration. L'hypothèse de départ en ADAJ était qu'une longue durée d'exécution d'une méthode se traduit dans un nombre important de méthodes invoquées. Un nombre important d'invocations est généré par des boucles. En conséquence, on supposera que les méthodes sans boucles sont courtes, donc elles ne nécessitent pas d'être arrêtées. L'avantage de ce choix est que les débuts de boucles sont faciles à repérer dans le bytecode, étant cibles des branchements en arrière.

En ADAJ, dans une première approche, d'une part, les méthodes statiques et du constructeur ne sont pas transformées ; d'autre part, les boucles dans les blocs *catch* et *finally* ne sont pas prises en compte, mais celles du bloc *try* le sont.

### Transformation d'une application

Une application ADAJ est pré-compilée par JavaParty, puis compilée par le compilateur Java et le compilateur RMI. Pour la transformation ADAJ, le bytecode d'une classe est analysé et seules les méthodes d'instance sont transformées pour insérer des points de migration.

Pour migrer un objet ayant une méthode en cours d'exécution, il faut pouvoir arrêter l'exécution de la méthode, sauvegarder le contexte d'exécution de la méthode, et lors du transfert des informations nécessaires, rétablir le contexte d'exécution et reprendre l'exécution de la méthode au point où elle avait été interrompue, conformément aux étapes décrites en 14.3.2.

Un attribut particulier, *etat*, indique l'état de l'objet, vis-à-vis d'une migration éventuelle. Si $etat = 1$, l'objet a reçu une demande de migration, si $etat = 0$, aucune demande de migration n'a été faite. L'ajout de cet attribut est particulièrement facile pour les fragments, mais moins évident pour tout autre objet remote. Pour de cette raison et celles spécifiées précédemment, les transformations suivantes correspondent aux fragments uniquement.

La transformation propose d'introduire des points de migration avant chaque boucle, où l'attribut *etat* sera testé. Si une demande de migration a été réalisée, l'état courant de l'exécution est sauvé, au travers d'un sous-programmes de type *sauver* et une méthode particulière, *migrer*, est appelée. Dans le cas contraire, la méthode continue à s'exécuter normalement, par un saut à l'instruction suivante. La méthode *migrer* indique que la migration de l'objet, en sens JavaParty, peut être exécutée. Pour cela, il faudra s'assurer que JavaParty puisse migrer l'objet, donc remettre le nombre d'invocations courantes à zéro.

Lors de l'arrivée de l'état des données et de l'état d'exécution sur la machine destinataire, il faut restaurer les données et la méthode ne peut pas être reprise depuis le début, car elle a pu changer l'état de l'objet courant ou d'autres objets. La méthode reprendra au point où elle était avant la migration. Ainsi, au début de la méthode, l'attribut *etat* est testé, si l'objet a migré, il faut :

- remettre à jour l'attribut *etat* (marquant que l'objet n'est plus demandé à migrer) et le nombre de méthodes courantes, en appelant la méthode *aMigre*,
- restaurer l'état d'exécution, en fonction du numéro de boucle à laquelle la méthode est arrivée, au travers d'un sous-programmes de type *restaurer* et
- sauter vers l'instruction suivante à exécuter, au travers d'une instruction de type *goto*.

Ces sauts nécessitent l'introduction d'un compteur simulant le compteur ordinal. Ainsi, l'attribut *no* de chaque fragment mémorise le numéro de la boucle.

### Exemple de transformation

Soit une classe de type fragment, ayant la méthode suivante, *m*, méthode qui comporte deux boucles imbriquées :

```
void m(int a, double b, String c) {
  int i, j, n = 10;
  ...
 for (i = 0; i < n; i++) { // boucle en i
      ...
      for (j = 0; j < n; j++) { // boucle en j
          ...
      }
      ...
  }
  ...
}
```

La transformation a la forme suivante, où les instructions ajoutées sont marquées en gras :

```
void m(int a, double b, String c) {
  int i, j, n = 10;
  if (etat == 0) goto deb;
  aMigre();
  switch (no) {
      case 0 : restaurer1; goto bcl1;
      case 1 : restaurer2; goto bcl2;
  }
  deb : ...
      for (i = 0; i < n; i++) { // boucle en i
          if (etat == 0) goto bcl1;
```

```
                sauver1() ; migrer() ;
  bcl1 : ...
                for (j = 0 ; j < n ; j++) { // boucle en j
                    if (etat == 0) goto bcl2 ;
                    sauver2() ; migrer() ;
  bcl2 : ...
                }
        }
}
```

### Sauvegarde et restauration de l'état d'exécution d'une méthode

La machine virtuelle Java est un interpréteur typé, fonctionnant avec une pile et un vecteur de variables locales pour chaque exécution de méthode.

Par exemple, pour la méthode d'instance

$$\text{m(int k, double d, String s)\{int i, j, n;}\cdots\}$$

et l'appel de méthode

$$x.m(a, b, c),$$

le vecteur de variables est initialisé de la manière suivante :

| this | a | b | b | c | | | |
|---|---|---|---|---|---|---|---|

où *this* est la référence vers l'instance courante ; $k$ étant un entier, $a$ prend une place ; $d$ étant un double, $b$ prend deux places ; et $s$ étant un objet, $c$ prend une place.

La taille de la pile est calculée par le compilateur, ainsi que le nombre de variables locales. La plupart des instructions prennent leur opérandes de la pile et remettent leur résultat dans la pile. Ainsi, pendant l'exécution d'une méthode, l'état des variables et la pile évoluent.

Pour pouvoir arrêter l'exécution d'une méthode et sauvegarder le contexte de son exécution, il faut connaître en tout point de la méthode, pour toutes les instructions, les types des variables et le contenu de la pile. Ceci se fait par un algorithme d'inférence de types, inspiré du "bytecode verifier", pour sa partie vérification de méthode, voir le détail en annexe E.

La sauvegarde des variables consiste à les ajouter, une par une, à un paquet de variables. Ce paquet, propre à chaque instance de fragment, est donc un attribut d'instance du fragment, instance d'une classe ADAJ : *fr.lifl.adaj.PaquetVars* (voir l'annexe E). Cette classe contient des méthodes pour ajouter tout type de variable (primitif : `double`, `float`, `int`, `long` ou référence : `Object`) et également des méthodes pour les récupérer.

Par exemple, si l'état du vecteur des variables est

| this | a | b | b | c |
|---|---|---|---|---|

où $a$, $b$, $c$ sont trois paramètres ci-dessous ($a$ un entier, $b$ un double et $c$ un objet), la sauvegarde est effectuée de la manière suivante :

- ignorer la référence sur l'objet courant, *this* (elle n'est pas sauvegardée),
- créer un paquet avec 3 variables,
- empiler $a$,
- empiler $b$,
- empiler $c$.

La procédure de sauvegarde ne peut pas être toujours la même (voir dans l'exemple de code précédent, où deux procédures de sauvegarde sont utilisées : **sauvegarde1** et **sauvegarde2**), parce que l'état de la pile peut être différent en fonction de la visibilité des variables ou des branchements du programme.

Dans l'exemple de code précédent, au début de la boucle en *i*, l'état des variables est

| this | a | b | b | c | i | ? | n |
|---|---|---|---|---|---|---|---|

où *j* n'a pas de valeur et il n'est pas possible d'accéder à *j* ; tandis qu'au début de la boucle en *j*, l'état est

| this | a | b | b | c | i | j | n |
|---|---|---|---|---|---|---|---|

Ou, dans le cas d'un branchement de type

$$\text{if (b)}\{\text{int i}\,;\cdots\}\ \text{else}\ \{\text{String s}\,;\cdots\}$$

l'état des variables sur la branche où la variable *b* est évaluée à vrai est

| this | b | i | ... |
|---|---|---|---|

et dans le cas contraire, l'état est

| this | b | s | ... |
|---|---|---|---|

Donc le même emplacement est utilisé pour mémoriser un entier ou une chaîne.

### Problèmes

Dans la version actuelle du mécanisme de migration en ADAJ, un certain nombre de cas où il y aurait pu avoir un point de migration ont été exclus. Ils ne sont pas considérés à cause du surcoût engendré par une transformation ou de l'impossibilité d'extraction de l'état complet de la pile.

Les différents cas non-traités sont :

- les méthodes statiques, qui ne peuvent pas être appelées sans instance,
- les constructeurs, qui sont appelés avec l'opérateur *new* et qui, suite à une migration, créeront de nouveau l'objet, au travers du même opérateur,
- les blocs *catch*, qui sont cibles de toute instruction du *try*, et leur transformation imposera un surcoût important,
- les blocs *finally*, qui posent un problème pour l'inférence de type, car les sous-programmes qu'ils appellent n'ont pas une source établie et les adresses de retour des sous-programmes sont impossibles à sauvegarder et restaurer,
- les méthodes sans boucles, mais qui appellent d'autres méthodes, qui ont des boucles (donc elles sont transformées), nécessitent une technique plus compliquée pour la sauvegarde et la restauration de la pile,
- plusieurs méthodes qui s'exécutent sur un objet peuvent être transformées pour sauvegarder leur contextes, mais il faudrait attendre que les deux méthodes passent par un point de migration, augmentant la latence de la migration.

### Intégration du mécanisme de migration ADAJ dans la plate-forme

La migration ADAJ est censée s'intégrer dans la plate-forme, sans modifier les fonctionnalités de migration JavaParty. Tenant compte que la migration JavaParty impose la contrainte que le nombre de méthodes courantes sur l'objet (*méthodes actives*) soit zéro, la migration ADAJ ne peut pas directement utiliser le même mécanisme. Deux transformations s'imposent, à part l'instrumentation du bytecode (qui permet d'accéder au contenu de la pile d'exécution), pour migrer en particulier les fragments avec une méthode active. Il s'agit de transformations :

- de la partie instance du fragment,
- du thread de lancement asynchrone d'une méthode de fragment (pour les appels distribute et asynchrones également).

Dans la partie instance d'un fragment, les deux opérations à définir concernent

- le déclenchement de la migration, dans le cas d'une seule méthode active, et
- la restauration de l'état de la variable de migration et du nombre de méthodes actives, lors de la fin de la migration.

La première opération sera exécutée par la méthode *migrer*, qui teste le nombre de méthodes actives, valeur fournie par un compteur de JavaParty. Si le nombre de méthodes actives n'est pas 1, la migration est abandonnée, en mettant l'indice de migration, la valeur de la variable *etat*, à zéro. Dans le cas contraire, le nombre de méthodes actives est décrémenté, de manière forcée, et une exception particulière de migration est levée, *MigrationException*. Cette exception transporte le numéro de la machine destinataire de la migration. L'opération concernant la mise à jour de la variable de migration, lors de la fin de la migration et du nombre de méthodes actives (par incrémentation), est réalisée par la méthode *aMigrer()*.

La deuxième transformation consiste à capter l'éventuelle exception levée par la méthode *migrer()* et réaliser la migration demandée. La prise en compte de cette situation est réalisée dans la classe du thread censé lancer le traitement asynchrone.

Si le lancement d'une méthode dans le thread est réalisé par la technique de réflexion (voir la solution dans la section 7.3), alors les exceptions levées par la méthode sont récupérées et relancées grâce à l'exception Java, *java.lang.reflect.InvocationTargetException*. Dans le cas où l'exception concerne la migration, la méthode de migration JavaParty est appelée, avec le paramètre transmis à l'aide de l'exception ADAJ *MigrationException*.

```
try {
   ...invoke(obj, param)...
} catch(InvocationTargetException e) {
   if (e.getTargetException() instanceof MigrationException ) {
      int dest = ((MigrationException)e).dest ;
      jp.lang.DistributedRutime.migrate(obj, dest) ;
   }
}
```

Si le lancement d'une méthode dans le thread est réalisé par la génération de code (voir la solution dans la section 7.5), alors la méthode *run()* du thread prévoit capter une nouvelle exception, *MigrationException*. Dans ce cas, la migration JavaParty est également engendrée, à l'aide de la même technique.

Dans les deux cas précédents, si une exception de migration est captée dans l'exécution du thread, le relancement de la méthode doit être prévu, pour qu'elle puisse continuer à s'exécuter dans la machine destinataire. Ce cas est prévu par un drapeau qui indique la nécessité de relancer la méthode.

Le coût de la migration ADAJ dépend du coût de la migration JavaParty, et y ajoute un surcoût lié à la sauvegarde et à la restauration de la pile, en fonction du type des éléments qu'elle contient. La transformation ADAJ augmente la taille du bytecode, en fonction du nombre de méthodes transformées et du nombre de points de migration. Pour un exemple particulier, le bytecode a été augmenté de 37,4 % par rapport au bytecode de la version observée.

## 14.4 Conclusions

La mise en œuvre du mécanisme d'équilibrage de charge est essentiellement basée sur la technique de migration. La propriété de migration des objets remote en JavaParty est exploitée en ADAJ, pour les objets globaux participant à la redistribution. La contrainte de migration passive est relâchée en ADAJ par l'introduction de la migration de fragments ayant une méthode active. La difficulté de la migration active réside dans l'extraction de la pile d'exécution d'une machine virtuelle Java.

Le problème de la migration des threads Java consiste dans l'extraction des informations liées à la pile d'exécution. Trois approches majeurs existent :

- par l'extension de la JVM (projet SIRAC),
- par la pré-compilation du code source (projet JavaGO),
- par la post-compilation du bytecode (projets Brake et JavaGOX).

ADAJ utilise la technique d'insertion du bytecode nécessaire à extraire, sauvegarder et restaurer l'état d'une pile Java. Cette solution a été proposée afin de conserver intacte la machine virtuelle Java, et également, d'utiliser la migration JavaParty, en l'étendant.

S. Bouchenak propose un mécanisme très efficace de migration, en dépit de la modification de la JVM. Une première version [Bou99] étendait aussi bien la machine virtuelle, que l'interpréteur, en gérant une pile de types, en parallèle avec la pile de la machine. Le surcoût constaté, dû à un double accès, pour les données, et la pile, a chaque instruction bytecode, a été éliminé dans une deuxième version, qui utilise uniquement l'extension de la machine. Cette méthode devient compatible avec le compilateur JIT[2], et les types sont récupérés au moment de la sérialisation du thread, par une déoptimisation dynamique JIT [BH02] (pour la version JDK1.3).

Notre approche conserve intacte la machine virtuelle Java. C'est une des premières contraintes imposées pour le développement d'un environnement d'exécution des applications distribuées. Des outils de bibliothèque, l'instrumentation du bytecode ou des mécanismes au niveau middleware offrent les informations nécessaires pour atteindre nos objectifs.

La migration des threads en ADAJ se rapproche ainsi de celle des projets JavaGOX ou Brake. Ces approches, au niveau de l'application, gagnent en portabilité mais perdent en efficacité et en extraction totale de la pile d'exécution (certains cas ne peuvent pas être traités).

Le prédécesseur de JavaGOX, JavaGO, propose de la migration de threads par pré-compilation du code source [SMY99]. Des structures de contrôle pour gérer les sauts à l'instruction suivante à exécuter sont difficiles à mettre en œuvre, à cause de l'absence des instructions de type *goto* dans le langage Java. La sauvegarde des frames pour les appels imbriqués utilise le mécanisme d'exception, qui nécessite une sauvegarde préalable, car la pile d'opérandes est vidée quand des exceptions sont levées. En JavaGOX [SSY00], l'extension de JavaGO, la technique de modification du bytecode est utilisée, en conservant, de JavaGO, la technique de transfert de l'état sérialisé d'un thread par un objet Java. Cela exige la modification des signatures des méthodes, procédé qui risque de poser des problèmes pour les applications utilisant de la réflexion, comme pour l'environnement ADAJ. Ainsi, ADAJ s'interdit la modification de la signature des méthodes, l'état sérialisé du thread étant géré par le fragment, grâce à un attribut d'instance.

Brake, dans un premier temps [TRC+00], réalise de la migration décidée, aussi comme JavaGOX, JavaGO ou SIRAC, étant désignée pour les systèmes à agents mobiles. Brake n'exige pas, au contraire, de modification des signatures des méthodes, grâce à l'association d'un contexte de sauvegarde à chaque thread. Les objets contexte sont gérés par un *manager de contexte* qui est capable de retrouver l'objet associé à un thread grâce à une clé de hachage associée au thread.

[2] Just-In-Time

Dans une extension [WTV02], proposée pour la prise en compte des threads distribuées (faisant des appels distants), les signatures des méthodes sont aussi modifiées, afin de transmettre un identificateur unique d'un thread distribué. La migration, dans ce cas, est forcée, mais une vérification de la demande de migration est testée à chaque appel de méthode.

En ADAJ, une inspection de la demande de migration est réalisée uniquement au début de chaque boucle. Une analyse plus fine des moments de migration est à effectuer, afin de réduire le surcoût lié à l'insertion du bytecode, et afin de réagir le plus vite possible à une demande externe de migration.

Une analyse comparative des projets cités précédemment qui proposent la migration des threads est donnée dans le tableau 14.2, par rapport au niveau d'implémentation, mécanisme de sauvegarde des frames, mode de migration (forcé ou décidé), modification de la signature des méthodes, type de thread migré (local ou distribué)[3].

| *caractéristiques* | SIRAC | Brake | JavaGO | JavaGOX | ADAJ |
|---|---|---|---|---|---|
| *niveau* | JVM | bytecode | code source | bytecode | bytecode |
| *sauvegarde frames* | pile // types (déoptim) | return/if | exceptions | exceptions | pas nécessaire |
| *mode initiative* | décidé | décidé (forcé) | décidé | décidé | forcé |
| *modif sign* | non | non (oui) | oui | oui | non |
| *type thread* | local | local (distr) | local | local | local |

Tableau 14.2 – Quelques systèmes existants proposant la migration des threads Java

En conclusion, la migration des threads exige soit la modification de la machine virtuelle (SIRAC), sans nécessiter de changement du code (ou bytecode) et offre un accès total à la pile Java, soit la modification au niveau application (du code, ou bytecode), qui nécessite des modifications de la signature des méthodes (JavaGO), ou une migration des objets actifs particuliers (comme les fragments, en ADAJ), ou des environnements particuliers gérant des contextes de threads (Brakes).

[3]pour les projets SIRAC et Brake, les caractéristiques de la version étendue sont entre parenthèses

# Chapitre 15

# Evaluation du mécanisme d'équilibrage ADAJ

Dans ce chapitre nous présentons les différents tests menés pour valider le mécanisme d'équilibrage de charge intra-application ADAJ. Les expérimentations se basent essentiellement sur l'application TSP (Travelling Salesman Problem) distribuée et parallèle implémentant l'algorithme en île décrit dans le chapitre 8. Nous avons choisi ce type régulier d'application pour la facilité d'observation de la qualité de la meilleure distribution à atteindre. Le paramètrage de l'application permet pourtant d'introduire une irrégularité de traitement. Un autre type d'application a été testé, pour analyser le comportement de l'algorithme d'équilibrage en présence de communication entre les objets.

La plate-forme d'exécution choisie comporte neuf machines homogènes, reliées par un réseau. L'intégration ultérieure de la charge des machines permettra d'étendre le mécanisme pour réagir également à des déséquilibres inter-applications, dans un environnement hétérogène.

Différentes caractéristiques d'un algorithme d'équilibrage de charge, comme l'efficacité, la capacité d'adaptation, le surcoût minimal ou la stabilité, ont été particulièrement analysées lors des évaluations.

## 15.1 Introduction

### 15.1.1 Propriétés d'un mécanisme de distribution de charge

Un mécanisme de distribution de charge est censé distribuer équitablement les charges des machines du système, afin d'améliorer la qualité d'une solution, exprimée fréquemment en temps d'exécution. Les propriétés désirables d'une stratégie de distribution de charge sont : l'efficacité, la capacité d'adaptation, le surcoût, la stabilité et la scalabilité.

**L'efficacité** mesure la qualité d'une solution obtenue, en appliquant la stratégie de distribution de charge par rapport à la qualité obtenue sans distribution. Une meilleure distribution peut améliorer les performances de l'exécution d'une application, pourvu qu'elle recouvre le coût généré par la décision de la distribution elle-même.

**La capacité d'adaptation** caractérise le degré auquel la stratégie s'adapte aux changements dans le système. Ces modifications interviennent lors de la disponibilité de nouvelles ressources, suite à l'ajout de nouvelles machines ou suite à la libération de ressources, ou lors de la disparition

subite de ressources précédemment disponibles, générée par la suppression des machines (volontaire ou involontaire, en cas de panne).

**Le surcoût** du mécanisme de distribution de charge dépend de deux facteurs : le coût des communications introduites par l'échange d'informations, la synchronisation, etc. et le coût processeur pour extraire la charge courante.

**La stabilité** caractérise le comportement de la stratégie de distribution dès lors que l'état auquel le système est arrivé n'engendre plus de redistribution. Distribuer les charges, dans un cas de stabilité, augmente le surcoût du mécanisme et risque de remettre en cause l'état précédent.

**La scalabilité ou le passage à l'échelle** (voir le chapitre 10) du mécanisme mesure le comportement de l'algorithme lors de l'augmentation du système, essentiellement en termes de nombre de machines.

### 15.1.2 Problèmes concernant les différentes stratégies de distribution de charge

Les principaux problèmes apparus lors de la considération d'un mécanisme de distribution de charge, et donc de celui en ADAJ, sont :

- *le surcoût introduit par le système de distribution de charge*
  Le système de distribution de charge lui-même produit une charge du système parce qu'il consomme des ressources pour collecter les informations de charge et pour réorganiser le système. Si la charge générée par le système de distribution est grande, alors l'efficacité obtenue grâce à ses décisions risque d'être recouverte. Ainsi, le surcoût devrait être maintenu le plus petit possible. Par exemple, les décisions optimales sont NP-complètes, et donc inadaptables pour usage dans un système de distribution de charge. Par contre, des heuristiques devraient être utilisées pour s'approcher des solutions optimales.
- *la sur-charge des machines légèrement chargées*
  Une machine momentanément légèrement chargée est préférable comme destination d'une entité à placer. Pour cela, il se peut que cette destination devienne sur-chargée dans le moment suivant parce que beaucoup d'entités ont été assignées sur la même machine, sans qu'une décision concertée soit prise. Ce processus peut être répété à l'infini et donc le système est caractérisé par une *instabilité*. Ce problème apparaît généralement dans les stratégies décentralisées, où les composants pour la distribution de charge ne coopèrent pas.
- *l'information de charge obsolète*
  Si l'état du système change rapidement et que les intervalles de temps pour la mise à jour de l'information sont trop longues, l'évaluation des charges risque de porter sur des informations obsolètes qui peuvent amener à de mauvaises décisions. En utilisant des intervalles plus courts, l'information est pertinente, mais le surcoût de la récupération de ces informations augmente. Un compromis entre le degré d'acceptation des valeurs de charge et le surcoût engendré devrait être fait.

## 15.2 L'environnement de test

Le mécanisme de redistribution d'objets implémenté en ADAJ fournit un équilibrage intra-application, en absence d'une information pertinente sur la charge d'une machine physique, ou sur le potentiel de la JVM. L'évaluation du mécanisme d'équilibrage de charge devrait donc

considérer un environnement d'exécution homogène, formé par des machines ayant les mêmes performances (car le calibrage des puissance des machines n'est pas réalisé). En outre, les machines ne doivent pas subir une autre charge à part celle de l'application elle-même (car la charge extérieure n'est pas directement prise en compte). De plus, l'interprétation des résultats serait facilitée si pour les applications testées, les situations idéales pourraient être estimées.

### 15.2.1 La plate-forme

La plate-forme d'exécution consiste en 7 machines de calcul, une machine pour le composant central (le *Runtime Manager*) et une machine pour le lancement du programme. Les machines sont homogènes, reliées par un réseau Ethernet, à 100 Mb/s. Les caractéristiques des machines sont : Pentium III, à une fréquence de 733 MHz, ayant 128 Mo de mémoire RAM[1]. Le système d'exploitation utilisé est Linux.

### 15.2.2 L'application TSP

Une première application testée est l'implémentation de l'algorithme parallèle et distribué qui résout le problème de voyageur de commerce. La solution fait appel aux techniques de la programmation génétique, l'algorithme intégral étant décrit dans le chapitre 8.

Le modèle d'exécution est du type BSP[2], avec une succession de *it* phases (appelées *itérations*) de : calcul parallèle, suivi d'une synchronisation et d'un échange cyclique de messages. Le calcul parallèle consiste à réaliser une opération d'évolution sur un nombre de sous-populations pendant un nombre de cycles, *lgcycle*. La taille d'une sous-population est un paramètre de l'algorithme, donné par la variable *taille*.

Différentes distributions des sous-populations ont été testées, en passant la configuration comme paramètre. Ces distributions sont résumées dans le tableau suivant, qui donne la répartition du nombre de sous-populations sur chacune des machines.

| distr \ n°mach | #0 | #1 | #2 | #3 | #4 | #5 | #6 |
|---|---|---|---|---|---|---|---|
| `d2.dat` | 0 | 15 | 20 | 5 | 30 | 15 | 30 |
| `ddes.dat` | 25 | 30 | 30 | 35 | 0 | 0 | 0 |
| `ddes2.dat` | 40 | 40 | 40 | 0 | 0 | 0 | 0 |
| `dseul.dat` | 70 | 0 | 0 | 0 | 0 | 0 | 0 |
| `de.dat` | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| `dequi.dat` | 17 | 17 | 17 | 17 | 17 | 17 | 18 |

Tableau 15.1 – Distributions initiales des sous-populations

En outre, des distributions aléatoires générées par la distribution initiale, par défaut, de JavaParty, ont été testées. Elle seront spécifiées dans la section 15.3.5.

Généralement, l'allure du lancement de l'application paramétrée est :

jadaj tspjpcgg.TspDistr [O|N] *fich-don taille lgcycle distr it*,

où `TspDistr` est l'application principale, du paquetage `tspjpcgg`, *fich-don* est le fichier contenant les villes et les connexions entre les villes et le paramètre optionnel O est une option pour l'affichage du meilleur chemin trouvé à l'itération courante.

[1]Les salles de TP, le soir, quand il n'y a plus personne, se sont révélées des situations idéales pour contrôler tous les facteurs, en particulier, les interférences avec les autres utilisateurs

[2]Bulk Synchronous Programming

Les différents algorithmes de distribution de charge testés prennent comme indicateur de charge le nombre de threads ($nbthr$) et la quantité de travail ($WP$) d'une machine virtuelle Java. Les différences interviennent pour les politiques de décision, qui sont les suivantes :

- algo20, où l'intervalle de normalité est définie par les limites suivantes :
  - $3 < nbthr_{norm} \leq nbthr_{moy} + nbthr_{ecrtmoy}$
  - $WP_{moy} - WP_{ecrtmoy} \leq WP_{norm} \leq WP_{moy} + WP_{ecrtmoy}$
- algo30 : algorithme de K-Moyennes et le ratio, entre l'écart moyen absolu et la moyenne, est comparé à un seuil
- algo40 : algorithme de K-Moyennes ; le ratio, entre l'écart moyen absolu et la moyenne, est comparé à un seuil ; et lissage des valeurs avec la dernière mesure,
- algo50 : algorithme de K-Moyennes ; le ratio, entre l'écart type et la moyenne, est comparé à un seuil ; et lissage des valeurs avec la dernière mesure.

Plusieurs valeurs du seuil, pour les algorithmes algo30, algo40 et algo50, ont été choisies : 0,088 ; 0,1 ; 0,3 ou 0,5.

De plus, le nom de la machine qui accueille le *Runtime Manager* est donné comme paramètre : *-host nommach*. L'allure d'un script de lancement se trouve dans l'annexe F.

### 15.2.3 L'application communicante

L'application TSP met en évidence des situations où la quantité de traitement à réaliser peut ne pas être uniformément distribuée, à cause des distributions inégales des sous-populations, ou à cause des tailles différentes de sous-populations à traiter. L'aspect communication n'intervient pas de manière importante, pour être pris en compte lors des décisions de migration.

Une deuxième application a été proposée afin de mettre en évidence l'importance de la prise en compte des communications. L'application est une simulation d'un algorithme pour résoudre des problèmes numériques par la méthode WR (Waveform Relaxation) itérative (voir l'application Medico Akzo Nobel [CWI]). Si la méthode directe trouve la solution exacte après un nombre fini d'opérations, la méthode itérative donne une solution approximative, après un nombre d'itérations, ayant une complexité moins importante.

Un algorithme itératif de type *waveform* consiste en quatre étapes :

- découpler le système en sous-systèmes,
- réaliser une estimation initiale de la solution,
- résoudre les sous-systèmes à l'aide d'une méthode conventionnelle, en utilisant la solution estimée d'un autre sous-système,
- itérer jusqu'à ce que la solution a convergé.

La version parallèle réalise le calcul de l'étape 2 en parallèle, avec un échange d'informations demandé par l'étape 3.

Le schéma d'un tel algorithme consiste dans des phases de calcul local, en parallèle sur différents processeurs, et d'échange d'informations avec d'autres processeurs.

En ADAJ, la simulation d'un tel algorithme considère les sous-systèmes des fragments d'une collection distribuée. Chaque fragment réalise une succession de communications vers le fragment voisin, suivie d'un calcul local. La communication consiste à demander un même type de calcul, comme celui local, mais prenant un temps d'exécution inférieur.

## 15.3 Description et interprétation des résultats des expérimentations

### 15.3.1 L'efficacité

L'efficacité d'un algorithme parallèle pour résoudre le problème TSP a été déjà mesurée dans la deuxième partie de la thèse, en se plaçant dans une situation idéale, qui ne nécessite pas d'apport du mécanisme d'équilibrage de charge. Dans cette partie, grâce au mécanisme de redistribution d'objets, nous nous intéressons à l'efficacité, en partant des situations déséquilibrées.

L'efficacité d'un algorithme d'équilibrage de charge est généralement mesurée en temps d'exécution. Le gain obtenu, en temps d'exécution, par rapport à une exécution sans équilibrage de charge, quantifie l'efficacité.

Le *gain relatif* est mesuré par la comparaison entre les temps d'exécutions des versions ADAJ observées sans que l'équilibrage de charge interagisse ($t_{obs+nolb}$), et des versions ADAJ observées avec le mécanisme d'équilibrage de charge redistribuant dynamiquement les objets ($t_{obs+lb}$).

$$gain_{rel} = \frac{t_{obs+lb} - t_{obs+nolb}}{t_{obs+nolb}}$$

Ce gain ne tient pas compte du surcoût engendré par l'observation. Ainsi, un *gain brut* est calculé en considérant le temps d'exécution de la même version ADAJ qui s'exécute avec l'équilibrage de charge, et le temps d'exécution de la version ADAJ non-observée et sans équilibrage de charge ($t_{noobs+nolb}$).

$$gain_b = \frac{t_{obs+lb} - t_{noobs+nolb}}{t_{noobs+nolb}}$$

Les deux types de gain ont été calculés pour les distributions suivantes : `d2.dat`, `ddes.dat`, `ddes2.dat` et `dseul.dat`, pour des sous-populations de taille 200 et 35 itérations. La fréquence de la politique d'information, qui caractérisait les moments d'inspection, était de 5000 ms. Les gains relatifs obtenus sont montrés dans le tableau 15.2. L'annexe G contient également les gains bruts obtenus.

Les résultats précédents sont présentés sous forme de graphe par la suite. Deux types de graphes sont présentés pour chaque couple (distribution, algorithme) : le premier démontre les gains obtenus en fonction de l'algorithme utilisé (les graphes à gauche des figures 15.1, 15.2, 15.3), et le deuxième démontre les gains obtenus en fonction des seuils choisis, pour les algorithmes algo30, algo40 et algo50 (les graphes à droite).

Pour les sous-populations de taille 200, et pour 35 itérations réalisées, les graphes montrent que le plus faible gain est obtenu, en général, par la stratégie algo20, à cause de l'instabilité du système. Ce comportement était anticipé par la remarque faite dans la section 13.3, qui montre qu'un déséquilibre est parfois détecté, même s'il n'est pas réel. Entre les algorithmes à seuil, pour un même algorithme (graphes à droite des figures), la tendance est l'obtention d'un gain de manière croissante jusqu'au seuil 0,3, mais on constate une diminution pour un seuil plus élevé à 0,5.

Le comportement change légèrement pour la distribution `dseul.dat`, où toutes les sous-populations sont concentrées au départ sur une seule machine. Dans ce cas, le meilleur gain obtenu est de 58,13 %, pour l'algorithme algo50, au seuil 0,1, sans, pourtant, que les gains obtenus pour le seuil 0,3 soient très mauvais. Les performances obtenues en utilisant le seuil 0,088 s'expliquent par la correction rapide du déséquilibre, qui couvrira le surcoût dû à l'instabilité observée précédemment.

Le gain, généralement, croît avec l'éparpillement de la distribution : le meilleur gain augmente de 17,05 % à 58,13 % pour les distributions `d2.dat`, `ddes.dat`, `ddes2.dat` et `dseul.dat`. Cette

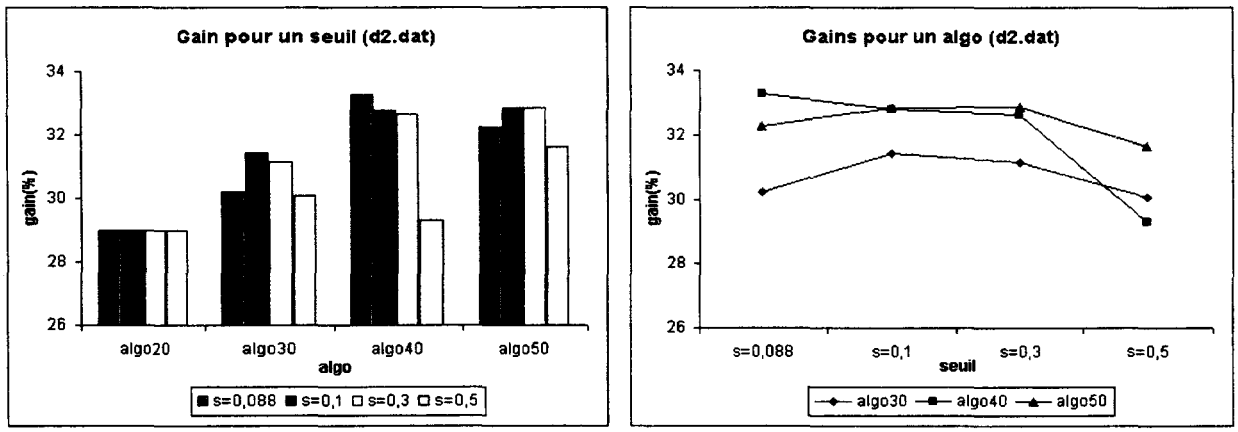



Figure 15.1 – Les gains relatifs obtenus pour la distribution d2.dat (taille 200, itérations 35)

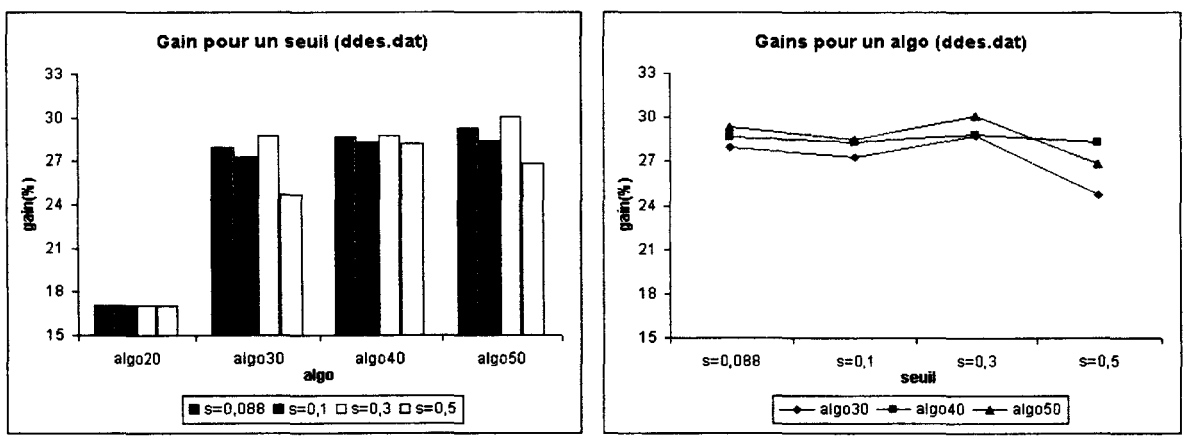



Figure 15.2 – Les gains relatifs obtenus pour la distribution ddes.dat (taille 200, itérations 35)

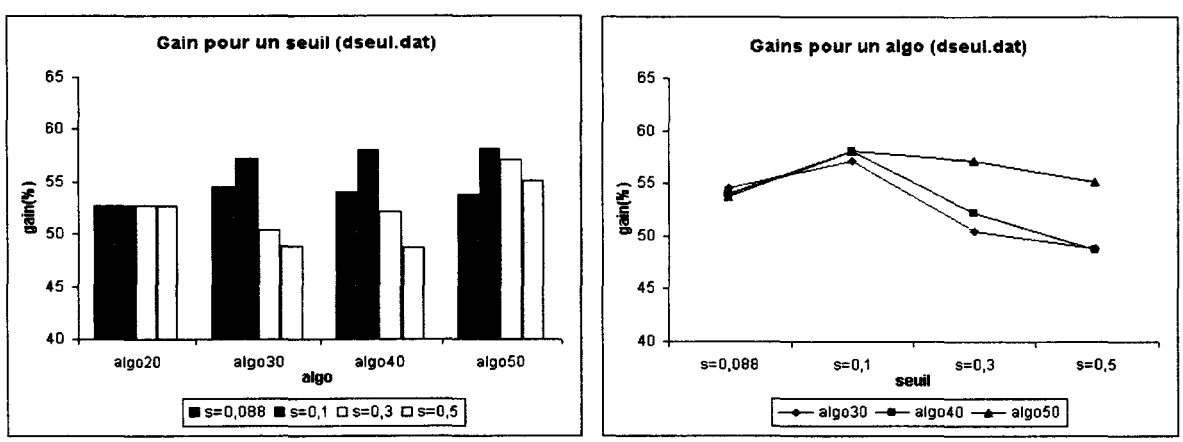



Figure 15.3 – Les gains relatifs obtenus pour la distribution dseul.dat (taille 200, itérations 35)

| $gain_{rel}$/ obs+nolb(%) | algo20 | seuil | algo30 | algo40 | algo50 |
|---|---|---|---|---|---|
| d2.dat | 29 | s=0,088 | 30,22 | 33,28 | 32,26 |
| | | s=0,1 | 31,43 | 32,78 | 32,84 |
| | | s=0,3 | 31,15 | 32,64 | 32,86 |
| | | s=0,5 | 30,08 | 29,31 | 31,62 |
| ddes.dat | 17,05 | s=0,088 | 27,96 | 28,67 | 29,3 |
| | | s=0,1 | 27,3 | 28,33 | 28,44 |
| | | s=0,3 | 28,73 | 28,79 | 30,08 |
| | | s=0,5 | 24,71 | 28,27 | 26,84 |
| ddes2.dat | 23,59 | s=0,088 | 28,63 | 31,38 | 31,19 |
| | | s=0,1 | 26,7 | 26,59 | 29,56 |
| | | s=0,3 | 23,3 | 28,73 | 29,82 |
| | | s=0,5 | 27,06 | 26,98 | 25,92 |
| dseul.dat | 52,69 | s=0,088 | 54,53 | 54 | 53,75 |
| | | s=0,1 | 57,17 | 58,1 | 58,13 |
| | | s=0,3 | 50,4 | 52,15 | 57,12 |
| | | s=0,5 | 48,82 | 48,7 | 55,13 |

Tableau 15.2 – Gains relatifs (en %) obtenus par les distributions de charge (taille 200, itérations 35)

croissance s'explique par le fait que la correction de charge initiale est importante et apporte plus de bénéfices que la correction d'une différence de charge initiale plus légère.

Les différentes distributions choisies s'étalent de celle à peu près uniforme (`d2.dat`) à celle complètement déséquilibrée (regroupement initial sur une seule machine : `dseul.dat`). Dans les applications réelles, où la distribution déséquilibrée n'est pas intentionnellement créée, les déséquilibres pouvant apparaître ne sont pas si forts, sauf dans le cas d'ajout ou de suppression de machines du système. Ainsi, par exemple, le comportement de l'application TSP pour la distribution initiale `dseul.dat` est caractéristique du cas d'un lancement de l'application sur une unique machine disponible dans le système, et ensuite, lors de la disponibilité de nouvelles ressources, par l'ajout de machines libres, l'application se déploie sur un nombre plus important de machines. Dans la perspective de déploiement d'application, c'est le cas de distribution d'une application dont l'exécution est centralisée au départ sur une seule machine.

### 15.3.2 Le surcoût

#### Le surcoût de l'observation

La différence entre les deux temps d'exécution, avec et sans observation, mais, dans les deux cas, sans équilibrage de charge, mesure le surcoût global engendré par l'observation, pour ce type d'application, en particulier.

$$surcoût_{obs} = \frac{t_{obs+nolb} - t_{noobs+nolb}}{t_{noobs+nolb}}$$

Lors des expérimentations concernant le problème TSP, le surcoût de l'observation a été calculé pour le test ayant comme paramètres d'entrée : la taille 200 d'une sous-population, le nombre de

cycles 30, le nombre d'itérations 35. Les résultats du tableau 15.3 montrent des surcoûts entre 0,07 % et 2,95 %.

Pour une autre application, [BOT02] détermine un surcoût maximal d'environ 5 %.

| paramètres | surcoût (%) |
|---|---|
| 200.35 - `d2.dat` | 2,78 |
| 200.35 - `ddes.dat` | 2,95 |
| 200.35 - `ddes2.dat` | 0,07 |
| 200.35 - `dseul.dat` | 1,83 |
| 200.20 - `de.dat` | 0,97 |

Tableau 15.3 – Surcoûts du mécanisme d'observation pour une application particulière

### Le surcoût du mécanisme de distribution

Le surcoût du mécanisme d'équilibrage de charge est généré par le contrôle du mécanisme de distribution, vu le temps pris par les différentes politiques. Le surcoût, généré par les opérations, locales ou distantes, est composé :

- du surcoût d'extraction d'information de charge (local),
- du surcoût de communication d'informations (distant), lié au temps pris par toutes les communications générées pour la diffusion des informations de charge,
- du surcoût de décision (local), lié à la détection d'un déséquilibre et à l'identification des machines sur/sous chargées,
- du surcoût de communication de la décision (distant),
- des surcoûts de sélection des candidats (locaux),
- des surcoûts de décision de placement (distants),
- des surcoûts de transfert des objets.

L'estimation de ces différents surcoûts individuellement n'est pas toujours possible. Ainsi, nous avons pu estimer le surcoût global du mécanisme d'équilibrage de charge pour une application dont le cas idéal (le cas qui donne la meilleure performance) est connu.

Deux types de tests ont été réalisés : pour une distribution identique (`de.dat`) et pour une distribution presque identique sur toutes les machines (`dequi.dat`). La première distribution initiale ne devrait pas détecter de déséquilibre, tandis que la deuxième risque d'engendrer des déséquilibres, détectés par certains algorithmes, donc de possibles migrations.

Le premier cas de test considéré est l'application TSP, avec une distribution initiale égale sur toutes les machines (`de.dat`) et exécutant 20 itérations. Un mécanisme d'équilibrage pour l'exécution de cette application dans un environnement homogène, non-perturbé par des charges extérieures n'est pas censé détecter des déséquilibres, donc il n'engendre pas de sélections locales de candidats, ni de migrations. Le surcoût qui peut être calculé, par la comparaison des temps d'exécutions de la version observée sans équilibrage et de la version observée avec équilibrage, est la somme des surcoûts de l'extraction de l'information, de communication de l'information de charge et de décision. Le surcoût de transfert est généralement proportionnel au surcoût de la migration d'un objet (en fonction du nombre d'objets migrés).

Le tableau 15.4 montre les surcoûts du mécanisme d'équilibrage en ADAJ, par rapport à une version observée du même algorithme, sans équilibrage. Les valeurs varient entre 1,71 % et 3,82 %, surcoûts qui mesurent uniquement le temps pris par l'extraction de l'information sur la charge, la

| surcoût/ obs+nolb(%) | algo20 | seuil | algo30 | algo40 | algo50 |
|---|---|---|---|---|---|
| de.dat | 3,29 | s=0,088 | 1,98 | 1,71 | 2,5 |
| | | s=0,1 | 2,63 | 3,03 | 1,71 |
| | | s=0,3 | 2,11 | 2,37 | 3,82 |
| | | s=0,5 | 1,98 | 2,24 | 3,03 |

Tableau 15.4 – Surcoûts des mécanismes de distribution de charge (taille 200, itérations 20)

diffusion de cette information et la politique de décision, car aucune migration n'a été demandée, sauf pour l'algorithme algo20 (3 déséquilibres ont été détectés).

Dans le deuxième cas (`dequi.dat`), pour 35 itérations, les surcoûts calculés sont plus importants (voir le tableau 15.5), vues les migrations engendrées. Les plus grands surcoûts sont obtenus pour l'algorithme algo20, à cause du nombre de migrations générées (28) et pour l'algorithme algo40, avec le seuil 0,088, pour la même cause (même si le nombre de migrations est inférieur, 14), mais aussi pour une mauvaise décision de placement, ce qui fait atteindre une étendue (l'écart entre la plus grande et la plus petite valeur) maximale de 4 pour le nombre d'objets sur les machines. Vue la synchronisation de tous les traitements entre deux itérations, une étendue importante ralentit considérablement la totalité du traitement (au plus lent des calculs). Les seuls algorithmes pour lesquels des migrations n'ont pas été engendrées sont les algorithmes algo40 et algo50, de seuils 0,3 et 0,5. Dans ces cas, le surcoût varie entre 3 % et 4,5 %.

| surcoût/ obs+nolb(%) | algo20 | seuil | algo30 | algo40 | algo50 |
|---|---|---|---|---|---|
| dequi.dat | 5,43 | s=0,088 | 3,65 | 7,01 | 2,35 |
| | | s=0,1 | 4,66 | 3,27 | 1,87 |
| | | s=0,3 | 1,78 | 3,27 | 3,12 |
| | | s=0,5 | 2,26 | 4,51 | 3,02 |

Tableau 15.5 – Surcoûts des mécanismes de distribution de charge (taille 200, itérations 35)

Dans le cas général, le surcoût du mécanisme dépend de l'état du système et des corrections imposées. L'intérêt d'un mécanisme de distribution est son intégration dans le framework d'exécution à un moindre coût. Plus le surcoût est important, plus le gain dû à la correction devrait l'être, pour couvrir le temps consommé par le mécanisme.

### 15.3.3 La stabilité

L'application TSP permet de suivre facilement un des objectifs de distribution de charge, celui d'atteindre une situation équilibrée. L'équilibre dans l'application de type TSP est atteint pour une distribution égale des sous-populations sur toutes les machines.

Un autre objectif d'un algorithme d'équilibrage est la stabilité ; c'est le souci de conserver un état d'équilibre, au moindre coût.

L'équilibre caractérisé, dans l'application TSP présentée précédemment, par un nombre équitable de sous-populations sur chaque machine, est maintenu par une quantité de travail et un nombre de threads quasi-égal sur toutes les machines. La quasi-égalité est caractérisée par l'appartenance des valeurs spécifiées ci-dessus à un intervalle autour de la moyenne, pour l'algorithme algo20 et à

une classe de valeurs autour de la moyenne, pour les algorithmes basés sur la méthode K-Moyennes (algo30, algo40, algo50). L'algorithme algo20 détecte presque toujours un déséquilibre entre les quantités de travail, tandis que les algorithmes K-Moyennes font un test préalable basé sur le seuil. Dans ces cas, le seuil est l'élément déclencheur du mécanisme de correction. Plus le seuil est petit, plus la distribution est équitable. La distribution est maintenue dans un *tuyau*, d'une largeur égale à l'étendue de la distribution, dépendante de la valeur de seuil.

La présence d'instabilité est reflétée dans les migrations à l'intérieur du tuyau. Elargir le tuyau, par une augmentation du seuil, peut éviter des migrations inutiles, donc des pertes de temps. En même temps, un tuyau plus large accepte des variations de charge importantes, donc des pertes de temps, générées par les moments de synchronisation. Les figures 15.4 et 15.5 montrent cet aspect pour la distribution initiale `ddes.dat` (l'algorithme appliqué est algo50), où les seuils de 0,088 et 0,1 rendent une distribution d'étendue 4, respectivement 5 (figure 15.4), mais avec des migrations inutiles dans le tuyau, tandis que pour un seuil de 0,3 ou 0,5, l'étendue est plus constante (4, respectivement 5, dans la figure 15.5), donc la stabilité est plus grande. Un compromis devrait être fait entre l'acceptation d'une variation maximale de charge, pour obtenir une meilleure performance, et le degré de stabilité, pour éviter des pertes en performances par des migrations inutiles. Maximiser le premier implique de rétrécir le tuyau, donc augmenter l'instabilité, et maximiser le deuxième implique d'élargir le tuyau, donc d'accepter un déséquilibre plus important de charge et en conséquence, des temps plus lents d'exécution.

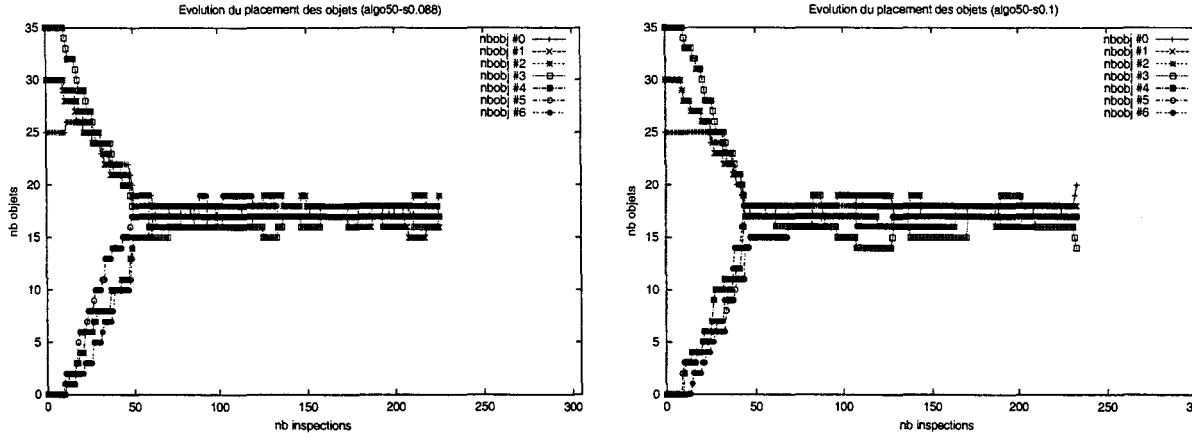



Figure 15.4 – Les évolutions des objets pour la distribution ddes.dat (seuils 0,088 et 0,1)

### 15.3.4 La scalabilité

La scalabilité d'un algorithme parallèle est la mesure de sa capacité d'utiliser un nombre de plus en plus important de processeurs. Cette capacité est directement liée à l'efficacité de l'algorithme, qui est mesurée par le ratio entre l'accélération et le nombre de processeurs.

Pour un problème de taille fixe (la même quantité de travail à effectuer), l'accélération d'un algorithme parallèle ne continue pas à augmenter avec le nombre de processeurs. Une saturation est acquise, formulée par la loi d'Amdahl, l'accélération étant bornée par l'inverse de la fraction séquentielle de l'algorithme. Cette saturation est due à l'augmentation des surcoûts avec le nombre de processeurs, ou à un excès de processeurs qui dépasse le degré de parallélisme de l'algorithme.

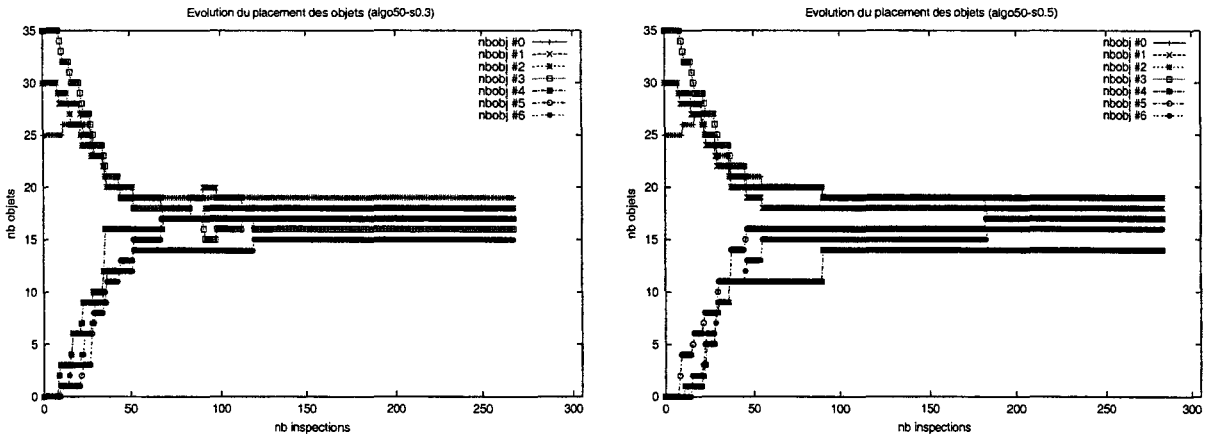



Figure 15.5 – Les évolutions des objets pour la distribution ddes.dat (seuils 0,3 et 0,5)

Deux scénarios initiaux sont envisageables pour un algorithme parallèle en présence d'un mécanisme d'équilibrage de charge :

- toute la quantité de travail est concentrée sur une seule machine,
- le travail est distribué équitablement sur l'ensemble de machines.

Dans le premier cas, pour une taille fixe du problème, l'efficacité diminue avec l'augmentation du nombre de machines. Cela s'explique par le fait que l'état d'équilibre est atteint, de plus en plus lentement, avec le nombre de machines, et l'équilibre est caractérisé par une granularité de plus en plus fine et un degré de plus en plus important. Ainsi, la partie "partiellement parallélisable" (avant l'état d'équilibre), est de plus en plus longue, avec l'augmentation du nombre de machines et donc l'accélération obtenue n'est pas linéaire avec le nombre de machines. Les expérimentations en ADAJ confirment ce résultat, par les efficacités obtenues (voir le tableau 15.6[3]) en déployant 120 sous-populations sur 2, 4 et 8 machines, populations initialement situées sur une seule machine (les machines concernées ont des processeurs Duron à 750 MHz, et 256 Mo RAM, reliées d'un réseau de débit 10 Mb/s).

| nb machines | temps d'exécution | efficacité |
|---|---|---|
| 2 | 134m16.306s | 0,58 |
| 4 | 104m11.907s | 0,37 |
| 8 | 84m42.201s | 0,23 |

Tableau 15.6 – Efficacités obtenues par le deployement de 120 sous-populations initialement placées sur une seule machine

Dans le deuxième cas, la partie d'amorçage, qui caractérise l'acquis de l'état équilibré, n'existe plus, ce qui permet d'obtenir de bonnes efficacités (voir le tableau 15.7).

Kumar est Rao [KG94] définissent une métrique de scalabilité, appelée *fonction d'isoefficacité.* Un système parallèle est appelé scalable, si l'efficacité peut être maintenue à une valeur fixe (entre 0 et 1) si le nombre de processeurs est augmenté, pourvu que la quantité de travail soit augmentée

[3]le temps d'exécution en séquentiel est 155m43.336s

| nb machines | temps d'exécution | efficacité |
|---|---|---|
| 2 | 112m50.929s | 0,68 |
| 4 | 64m48.270s | 0,60 |
| 8 | 38m8.248s | 0,51 |

Tableau 15.7 – Efficacités obtenues par le deployement de 120 sous-populations équitablement sur le réseau

également. Le type d'incrémentation de la quantité de travail, linéaire, ou exponentielle, avec le nombre de processeurs, rend le système grandement scalable, ou respectivement faiblement scalable.

### 15.3.5 L'intérêt

Un autre test démontre que même une distribution aléatoire initiale, telle que celle générée par JavaParty, peut être améliorée, par le mécanisme d'équilibrage de charge en ADAJ. Ainsi, la même application TSP a été testée, avec, comme données d'entrée, des sous-populations (variant par la taille), la longueur de cycle et le nombre d'itérations. La distribution aléatoire générée par JavaParty sera sauvegardée dans un fichier de distribution. Il sera utilisé par la suite, par cette application TSP, qui partira de cette distribution. Trois fichiers de distribution ont été testés : deux, pour des sous-populations de 200 et un, pour des sous-populations de 500. Les distributions initiales, présentées dans le tableau 15.8, donnent le nombre de sous-populations par machine virtuelle.

| distr \ n°mach | #0 | #1 | #2 | #3 | #4 | #5 | #6 |
|---|---|---|---|---|---|---|---|
| `djp1.dat` | 16 | 14 | 15 | 20 | 18 | 22 | 15 |
| `djp2.dat` | 12 | 22 | 19 | 19 | 12 | 14 | 22 |
| `djp3.dat` | 21 | 17 | 12 | 12 | 28 | 12 | 18 |

Tableau 15.8 – Distributions aléatoires des sous-populations, générées en JavaParty

Les temps comparatifs des exécutions des versions JavaParty et ADAJ, pour la même distribution initiale, aléatoire, montrent des gains obtenus, grâce au mécanisme de distribution. Ce gain croît avec l'augmentation de la taille des grains du traitement (voir le tableau 15.9).

| paramètres | $gain_b$(%) | distribution finale | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | #0 | #1 | #2 | #3 | #4 | #5 | #6 |
| 200.35 - `djp1.dat` | 8,58 | 16 | 18 | 18 | 17 | 17 | 17 | 17 |
| 200.35 - `djp2.dat` | 4,67 | 19 | 15 | 17 | 18 | 18 | 18 | 15 |
| 500.20 - `djp3.dat` | 23,32 | 17 | 16 | 18 | 19 | 14 | 17 | 19 |

Tableau 15.9 – Gains bruts pour les distributions aléatoires générées en JavaParty

### 15.3.6 La réactivité de l'algorithme face aux situations irrégulières

L'application TSP testée précédemment a l'avantage d'être facile à observer, par la connaissance préalable de la meilleure situation à atteindre. En même temps, elle a l'inconvénient d'être trop régulière et donc de ne pas présenter des aspects variés dans l'exécution. L'irrégularité d'une application se caractérise par des créations dynamiques d'objets ou par des variations dans les différentes

phases de calcul. L'irrégularité testée a été introduite au niveau de la quantité de travail exécutée par chaque objet. Cette application, TSP, a été transformée, pour accepter des sous-populations de tailles variées. La distribution initiale des sous-populations est bien équilibrée en nombre de sous-populations (`de.dat`). Les sous-populations placées sur les machines ont la taille donnée par le tableau 15.10.

| taille \ n°mach | #0 | #1 | #2 | #3 | #4 | #5 | #6 |
|---|---|---|---|---|---|---|---|
| `taille1.dat` | 50 | 100 | 50 | 25 | 200 | 40 | 20 |
| `taille2.dat` | 50 | 100 | 200 | 250 | 300 | 400 | 350 |

Tableau 15.10 – Tailles des sous-populations pour chacune des machines

Les deux granularités, fine (`taille1`) et moyenne (`taille2`), mettent en évidence des quantités de travail différentes, dans le contexte d'un même nombre de threads par machine.

Les mesures d'évaluation sont à la fois le temps d'exécution des versions observées, avec et sans équilibrage de charge, et le nombre d'individus traités par machine. La première mesure donne le gain obtenu par un mécanisme d'équilibrage, tandis que la deuxième analyse les cas d'équilibre atteints : un nombre quasi-égal d'individus pour chaque machine caractérisant ces situations équilibrées.

Les temps d'exécution, ainsi que les gains obtenus, sont présentés dans le tableau 15.11.

| paramètres | temps (obs+lb) | temps (obs+nolb) | $\text{gain}_{\text{rel}}$ (%) |
|---|---|---|---|
| 20 it - `taille1.dat` | 10m58.878s | 12m16.158s | 10,6 |
| 35 it - `taille1.dat` | 17m48.119s | 20m53.018s | 14,76 |
| 20 it - `taille2.dat` | 19m3.449s | 23m17.498s | 18,18 |
| 35 it - `taille2.dat` | 32m49.859s | 41m16.448s | 20,47 |

Tableau 15.11 – Temps d'exécution et gains relatifs obtenus pour une application irrégulière en granularité de traitement

Le mécanisme d'équilibrage de charge détecte des déséquilibres et équilibre la charge par des migrations d'objets. Vue la variété des granularités de traitement pour les objets, le nombre d'objets ne convient plus comme mesure d'évaluation d'un équilibre. Dans ce cas, il a été remplacé par le nombre d'individus à traiter par machine. Il mesure significativement la qualité de distribution de la charge. Les figures 15.6 et 15.7 montrent les granularités de traitements par machine, pour les deux fichiers contenant les tailles (`taille1.dat` et `taille2.dat`), et un nombre de 20, respectivement 35 itérations.

Etant donnée la configuration de départ, la situation idéale est atteinte pour un nombre approximatif de 689 individus, pour la granularité fine, et de 2357 individus pour la granularité moyenne, à traiter par machine. Les tests montrent une meilleure adaptabilité du système, pour la granularité moyenne, tandis que, pour la granularité fine, l'équilibre est atteint plus difficilement.

Le gain obtenu croît naturellement avec le nombre d'itérations (de 10,6 % à 14,76 % pour `taille1` ou de 18,18 % à 20,47 % pour `taille2`), et avec l'augmentation de la granularité des traitements (de 14,76 % à 20,47 %).

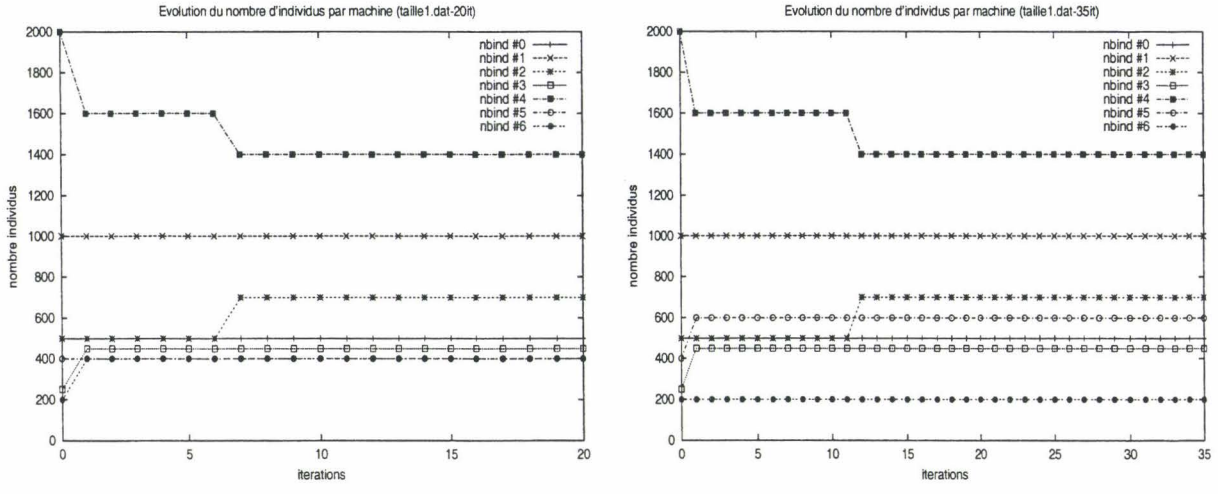



Figure 15.6 – L'évolution de la granularité de traitement par machine pour des sous-populations de granularité fine (20 et 35 itérations)

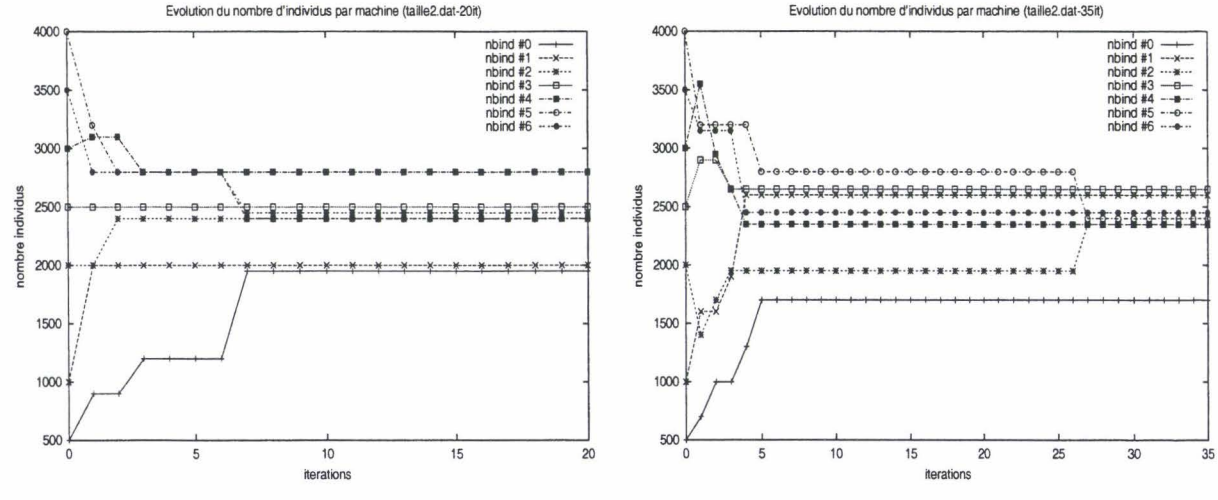
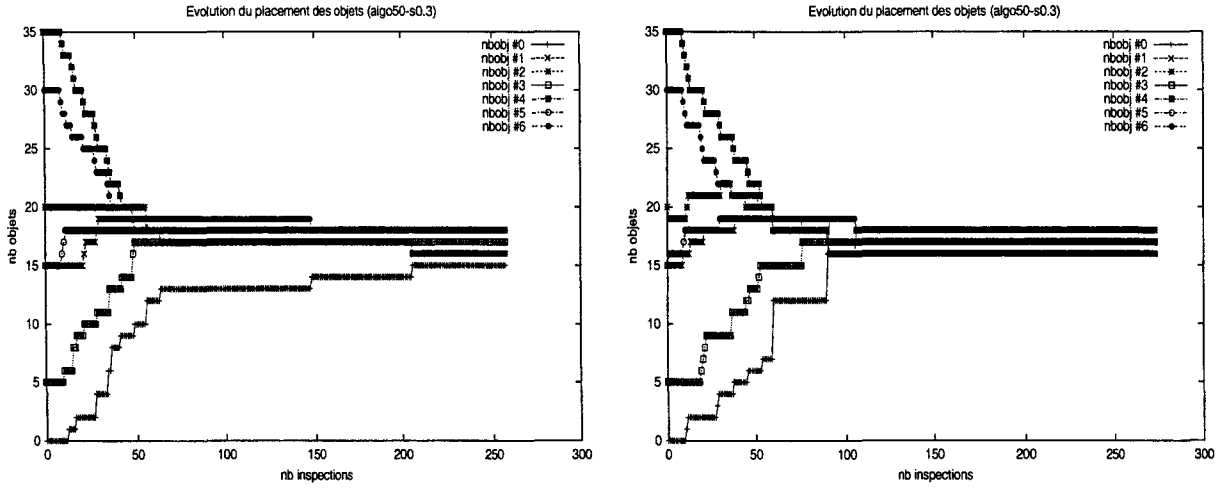



Figure 15.7 – L'évolution de la granularité de traitement par machine pour des sous-populations de granularité moyenne (20 et 35 itérations)

### 15.3.7 Migration ADAJ versus migration JavaParty

La migration ADAJ, présentée dans la section 14.3.4, relâche la contrainte imposée par le mécanisme JavaParty, sur l'état des objets migrables. Ainsi, des objets, avec une seule méthode en cours d'exécution, peuvent migrer. L'apport de la migration ADAJ, par rapport à la migration JavaParty, est illustrée grâce à la même application TSP, dans le contexte d'une distribution initiale déséquilibrée. Le moyen de transfert est la migration JavaParty, et en cas d'échec, la migration ADAJ. Les paramètres de l'application sont : la taille (200) d'une sous-population, la longueur (30) du cycle et le nombre (35) d'itérations ; les paramètres de l'exécutif sont : l'algorithme `algo50`, le seuil : 0,3, la fréquence d'inspection : 8 sec et la période de lissage : 5 sec.




(a) Mécanisme JavaParty

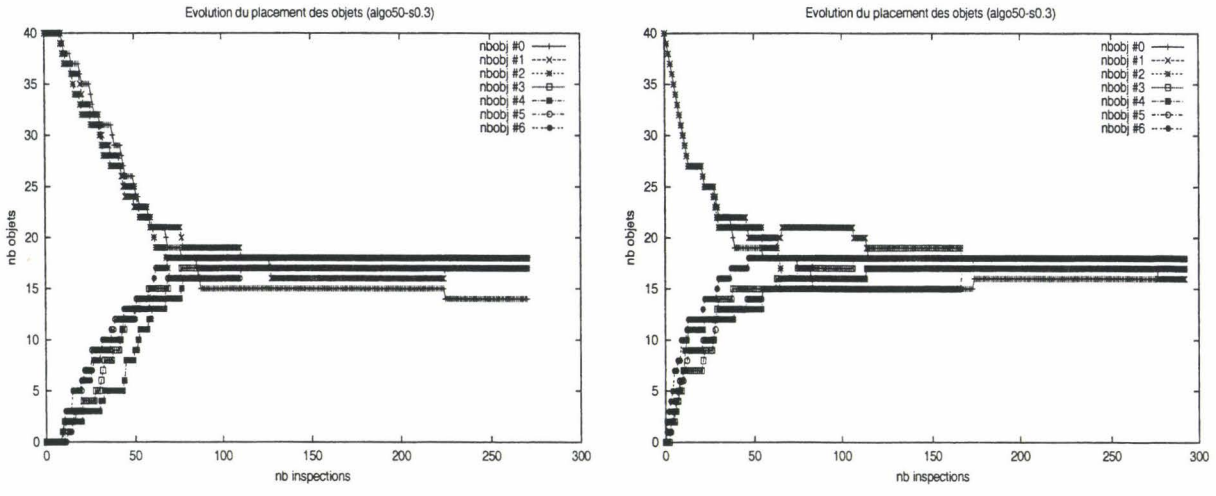
(b) Mécanisme ADAJ

Figure 15.8 – Comparaison des mécanismes de transfert JavaParty et ADAJ pour la distribution initiale d2.dat

Le comportement des objets, en termes de localisation, est montré dans les figures 15.8, 15.9 et 15.10, respectivement, pour les distributions `d2.dat`, `ddes2.dat` et `dseul.dat`. La nature de l'application permet des migrations ADAJ uniquement dans la phase d'initialisation des sous-populations. Donc, plus cette phase est longue, plus la possibilité de migration des objets avec une méthode en cours d'exécution est grande. Cet aspect est reflété aussi dans les graphes, comme dans les temps d'exécution. Pour la distribution initiale `d2.dat`, la différence par rapport à l'utilisation du mécanisme JavaParty de transfert n'est pas nette, à cause du temps réduit pris par l'initialisation. Pour les distributions `ddes2.dat` et `dseul.dat`, les migrations des objets depuis le début de l'application sont visibles, et apportent une amélioration dans le temps d'exécution (voir le tableau 15.12).

### 15.3.8 Rôle des communications

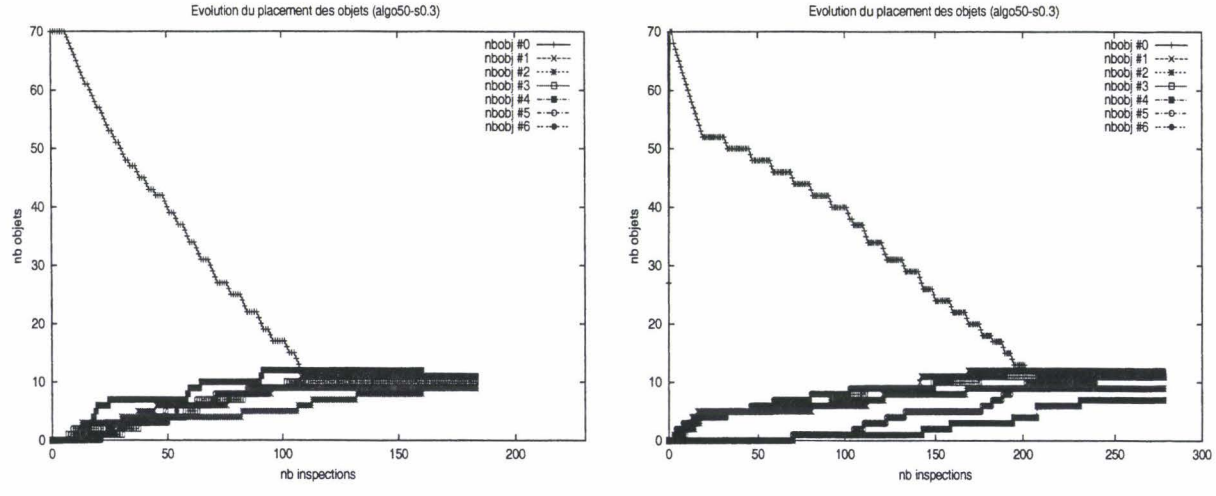
Cette section porte sur l'analyse du comportement de la deuxième application testée, en vue de la prise en compte des communications.




(a) Mécanisme JavaParty

(b) Mécanisme ADAJ

Figure 15.9 – Comparaison des mécanismes de transfert JavaParty et ADAJ pour la distribution initiale ddes2.dat




(a) Mécanisme JavaParty

(b) Mécanisme ADAJ

Figure 15.10 – Comparaison des mécanismes de transfert JavaParty et ADAJ pour la distribution initiale dseul.dat

| paramètres | temps (obs+lb) migration JavaParty | temps (obs+lb) migration ADAJ | $gain_b$ (%) |
|---|---|---|---|
| d2.dat | 40m35.588s | 40m3.051s | 1,31 |
| ddes2.dat | 45m13.661s | 42m49.641s | 5,31 |
| dseul.dat | 47m35.381s | 44m2.669s | 7,46 |

Tableau 15.12 – Temps d'exécution et gains obtenus avec la technique de migration ADAJ

### Analyse du comportement de l'application communicante

Le schéma d'exécution de l'application communicante, présenté dans la figure 15.11, montre que chaque fragment exécute, simultanément, un calcul demandé par le fragment précédent, et un traitement local (sauf pour le dernier fragment qui n'exécute pas une communication, et pour le premier qui ne reçoit pas de calcul à exécuter). Le traitement local est interrompu pendant la communication, parce que l'appel vers le fragment voisin est bloquant. Ainsi, à chaque itération, il y a au plus deux méthodes actives et au moins une méthode active (lors de la fin de calcul issu de la communication).

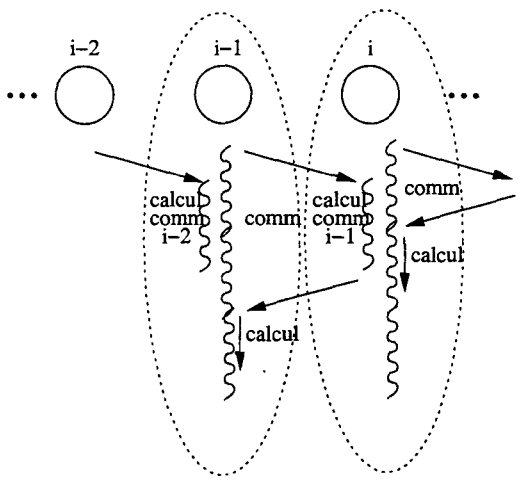



Figure 15.11 – Le squelette de l'application communicante

Dans ces circonstances, l'outil de migration JavaParty s'avère difficilement exploitable, vu le nombre de méthodes actives. Au contraire, le mécanisme de migration ADAJ peut apporter de bons résultats, notamment lors de la phase de calcul propre au fragment.

La prise en compte des communications intervient dans la politique de localisation. Le mécanisme d'équilibrage de charge en ADAJ vise la distribution équitable de la charge, en vue d'optimisation des communications, et ne réagit pas à un déséquilibre de communications. Ainsi, la distribution initiale des fragments est volontairement déséquilibrée et réalisée de telle façon que les communications ne soient plus en anneau. Par exemple, si les fragments sont numérotés de 0 à 12, leur distribution initiale sur 4 machines est montrée dans la figure 15.12.

Cette distribution génère 10 *communications externes* (entre des objets globaux sur des machines différentes, en arc plein dans la figure) et 2 *communications internes* (entre des objets globaux sur la même machine, en arc pointillé dans la figure). En réagissant au déséquilibre de charge généré par cette distribution, le mécanisme de redistribution est censé corriger la distribution d'objets, tout en choisissant des placements qui n'engendrent pas plus de "mauvaises communications" (externes),

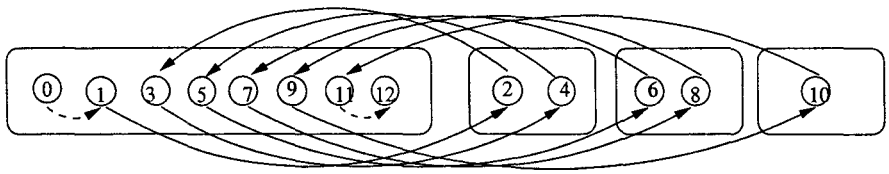



Figure 15.12 – Distribution initiale des fragments pour l'application communicante

mais au contraire, les diminuent.

### Intérêt de rapprocher deux objets globaux

L'objectif de la prise en compte des communications lors de la décision sur le placement d'un objet migré est de remplacer des communications externes, par des communications internes. Même si une communication interne est toujours distante (avec une sérialisation et désérialisation d'objet), éliminer le transfert des informations à travers le réseau améliore le temps d'exécution. Cela a été validé par un test qui réalise un nombre différent d'invocations ($n$) entre deux objets remote. Les machines qui sont intervenues sont homogènes : Pentium III, à une fréquence de 733 MHz, ayant 128 Mo de mémoire RAM, reliées d'un réseau à débit de 100 Mb/s. Le tableau 15.13 montre un ralentissement moyen de 22 % pour une communication externe, par rapport à une communication interne[4].

| | temps comm. int. | temps comm. ext. | surcoût (%) |
|---|---|---|---|
| n = 500 | 117 ms | 140 ms | 19,65 |
| n = 1000 | 224,66 ms | 275,33 ms | 22,55 |
| n = 5000 | 1094 ms | 1366 ms | 24,86 |
| n = 10000 | 2217,66 ms | 2728,66 ms | 23,04 |

Tableau 15.13 – Temps d'exécution et surcoûts des communications externes par rapport à des communications internes

Le débit réseau joue un rôle important dans ces mesures : un trafic moins rapide ralentit encore plus les communications externes.

La pénalité des communications externes est d'autant plus importante que des optimisations de communications peuvent être introduites : des objets remote qui se situent sur la même machine communiquent localement et dans ces cas, une communication externe peut effectivement devenir une communication locale (voir la version 0.98 de JavaParty).

### Résultats

L'objectif des tests effectués concernant l'application communicante est de mettre en évidence le comportement du mécanisme d'équilibrage de charge en présence des communications. Les choix interviennent lors de la politique de localisation, d'où l'intérêt d'analyser une distribution d'objets finale, pour différentes valeurs du paramètre $\alpha_{attre}$ (coefficient de pondération de l'attraction externe par rapport au travail de la JVM, dans la politique de localisation) :

- $\alpha_{attre} = 0,4$,

[4]les temps d'exécution sont des moyennes de 5 exécutions, dont le meilleur et le pire des temps ont été éliminés

- $\alpha_{attre} = 1$,
- $\alpha_{attre} = 0$.

Le premier cas correspond à une pondération presque égale de l'attraction extérieure (traduite en communications externes) et la quantité de travail de la machine destinataire, en préférant pourtant la deuxième des mesures. Deux des distributions finales sont montrées dans la figure 15.13. A part le niveau de charge équilibré atteint, les communications externes sont diminuées, à 6 en moyenne, par rapport aux 10 initiales.

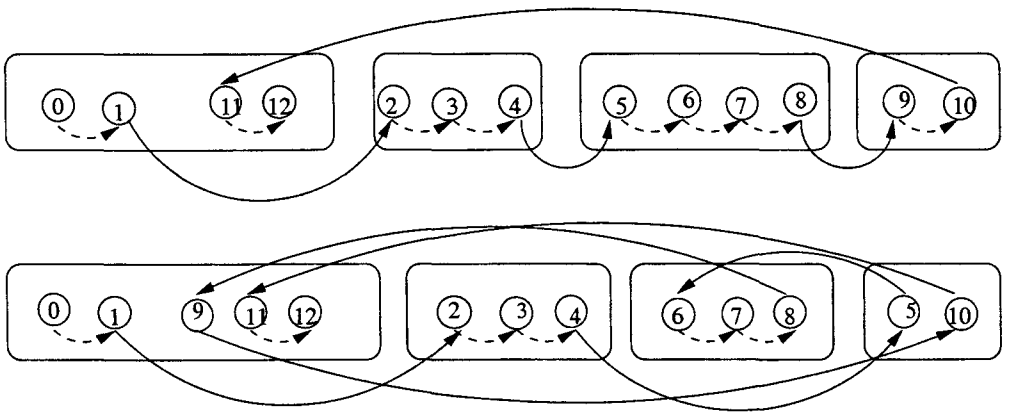



Figure 15.13 – Distributions finales des fragments pour l'application communicante ($\alpha_{attre} = 0,4$)

Pour $\alpha_{attre} = 1$ les tests ont montrés une très légère différence par rapport au cas précédent. Ici, les communications sont préférées, à 100%, c'est-à-dire elle déterminent, en fonction de leur intensité, la machine destinataire de l'objet. Les distributions finales, ainsi que les communications externes sont satisfaisantes, même sans considérer la quantité de travail des machines destinataires, parce que celles-ci sont déjà choisies parmi les sous-chargées.

Dans un dernier cas, les communications ne sont pas prises en compte ($\alpha_{attre} = 0$). C'est le cas d'un algorithme d'équilibrage qui néglige l'aspect communication. L'équilibrage de charge est réalisé, comme le montre la figure 15.14, mais il n'y a pas d'amélioration des communications externes (restent à 10). De plus, pour d'autres exécutions, le hasard du choix de la machine a augmenté le nombre de communications externes.

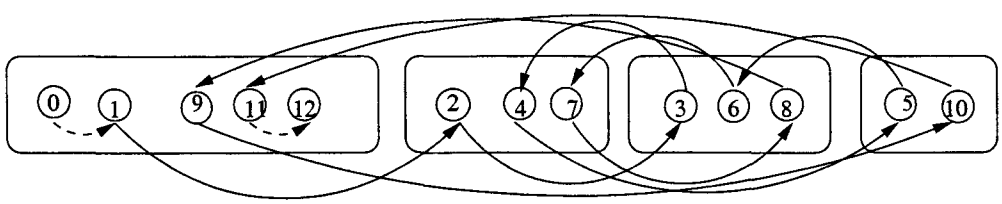



Figure 15.14 – Distribution finale des fragments pour l'application communicante ($\alpha_{attre} = 0$)

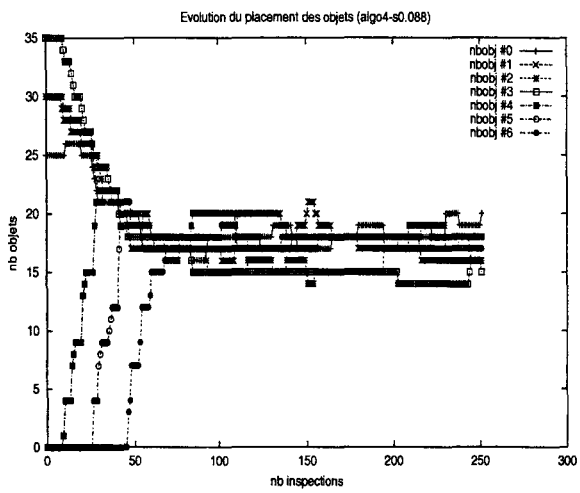
Les tests menés sur l'application communicante ont mis en évidence l'importance de la considération des communications lors du mécanisme d'équilibrage en ADAJ.

## 15.4 Discussions

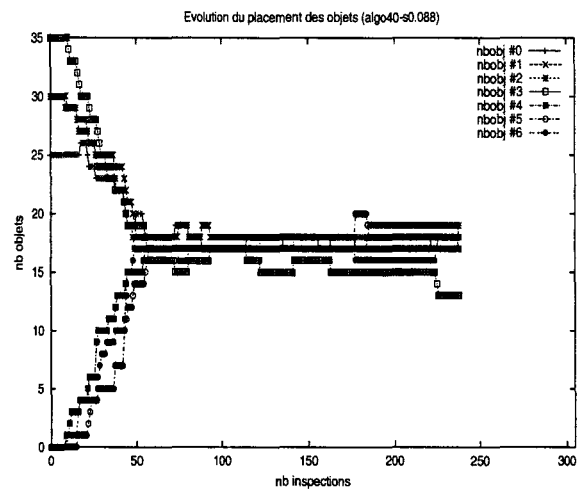
D'autres aspects particuliers du mécanisme d'équilibrage de charge ont été analysés, comme l'influence de la fréquence d'inspection, le choix d'une machine destinataire, ou le comportement des mesures de charge dans une situation de charge externe. Ici, nous détaillons uniquement les deux derniers aspects. Une redistribution éventuelle des objets (en cas de déséquilibre) est dépendante des moments d'inspection, quand la remonté des informations de charge est réalisée. Une remontée très fréquente a l'avantage de disposer d'une charge récente des machines, mais risque de générer un surcoût considérable, par des communications et des extractions de la charge. A l'inverse, si la remontée est peu fréquente, les informations de charge analysées seront obsolètes et les décisions risquent d'être inadéquates.

### 15.4.1 Choix de la machine destinataire

Le choix de la machine destinataire est réalisé en fonction des critères d'attraction et de minimisation du travail d'une JVM. Des expérimentations ont montré que, dans différents cas, il peut y avoir des effets de congestion d'une seule machine, à cause de la prise en compte des machines sous-chargées avec la même importance, par les machines sur-chargées. Alors, la politique de placement de chaque machine sur-chargée décide de placer ses objets sur une même machine sous-chargée. Un exemple est montré dans la figure 15.15(a), pour l'exécution de l'application TSP.




(a) L'effet de congestion de la politique de placement




(b) Politique de placement considérant les machines sous-chargées en ordre aléatoire

Figure 15.15 – Choix de la machine destinataire

Pour éviter cet aspect, en ADAJ, les machines sous-chargées sont considérées dans un ordre aléatoire par toute machine sur-chargée, ceci en plus de la prise en compte des critères de localisation. Ceci, pour donner une chance à chaque machine sous-chargée d'être choisie comme destination d'un objet (voir la figure 15.15(b)).

### 15.4.2 Equilibrage de charge inter-applications

L'extension de l'algorithme d'équilibrage de charge ADAJ, pour prendre en compte les charges extérieures à l'application ADAJ, générées par d'autres applications, permet l'intégration d'un algorithme d'équilibrage inter-applications. Les déséquilibres de charge à l'intérieur d'une application sont reflétés dans les métriques de charge utilisées, dans un système dédié. La question qui se pose, lors d'un mécanisme d'équilibrage inter-applications, est : est-ce que ces métriques reflètent aussi une charge extérieure à l'application ?

Le nombre de threads actifs ne donne pas une information intéressante pour cette question. C'est une mesure constante, indépendante du partage du processeur entre plusieurs processus. Le remplacement de la mesure du nombre de threads par le nombre de processus prêts à s'exécuter sur la machine pourrait donner une information plus pertinente sur la charge globale de la machine. L'impossibilité d'avoir un outil, compatible Java, pour obtenir la longueur de la queue CPU (mesure de charge utilisée dans d'autres systèmes) nous a amené à implémenter un outil Java d'estimation de la charge d'une machine physique [BOT01] : il mesure le temps d'attente lors du changement de contexte entre les processus. Ce mécanisme se base sur l'ordonnancement des threads ; il est dépendant, donc, de la plate-forme (voir la section 12.1.2).

La deuxième mesure de charge utilisée par l'algorithme d'équilibrage intra-application est la quantité de travail d'une machine virtuelle. Cette mesure reflète l'existence d'autres processus prêts à s'exécuter sur une machine : le nombre de processus prêts est d'autant plus grand, que le temps CPU, affecté pour le processus de la JVM, est réduit. Donc, la quantité de travail, pouvant être exécutée dans un même intervalle de temps, est d'autant plus réduite.

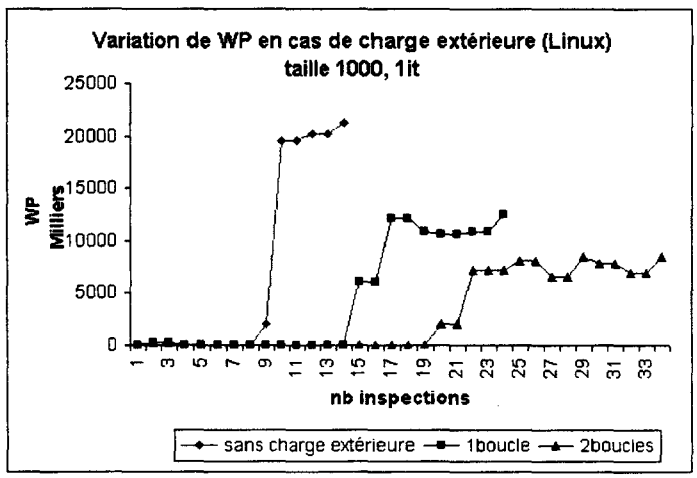
Cette conclusion a été validée par une expérimentation menée sur deux systèmes différents (Solaris et Linux). Une seule machine de calcul a été analysée, par rapport aux valeurs des quantités de travail de la JVM (voir la figure 15.16). La machine tournait une application ADAJ, dans le contexte d'une charge extérieure générée par un programme demandant du temps CPU, appelé `boucle` (1, respectivement 2 executions simultanées ont été effectuées pour générer de la charge extérieure).

## 15.5 Conclusions

Dans ce chapitre, l'évaluation du mécanisme d'équilibrage en ADAJ est présentée. Deux types d'applications ont été testés : une première orientée calcul (TSP) et une deuxième qui met en évidence des communications.

Les résultats montrent l'obtention des gains considérables, par rapport aux versions qui sont exécutées sans équilibrage de charge. Ces gains sont mesurés en temps d'exécution. Un deuxième objectif analysé était la stabilité du mécanisme : dès que l'état arrive à un équilibre, et des perturbations n'interviennent plus, le système est censé rester stable et ne pas engendrer des migrations inutiles.

Les analyses portent sur les différentes politiques de décision d'un déséquilibre, avec différents seuils. L'algorithme algo20 est généralement le moins efficace à cause de l'instabilité des localisations des objets dans le tuyau : des situations de déséquilibre sont détectées parce que dans l'algorithme autour de la moyenne, des valeurs existent fréquemment en dehors de l'intervalle de normalité. La différence entre les algorithmes algo30, algo40 et algo50, basés sur l'algorithme KMoyenne de classification, n'est pas essentielle, résultat confirmé par les gains obtenus. La considération du coefficient de variation, au lieu du rapport entre l'écart moyen et la moyenne, pour les valeurs de charge, pénalise plus l'hétérogénéité des valeurs. Une correction rapide d'un déséquilibre important



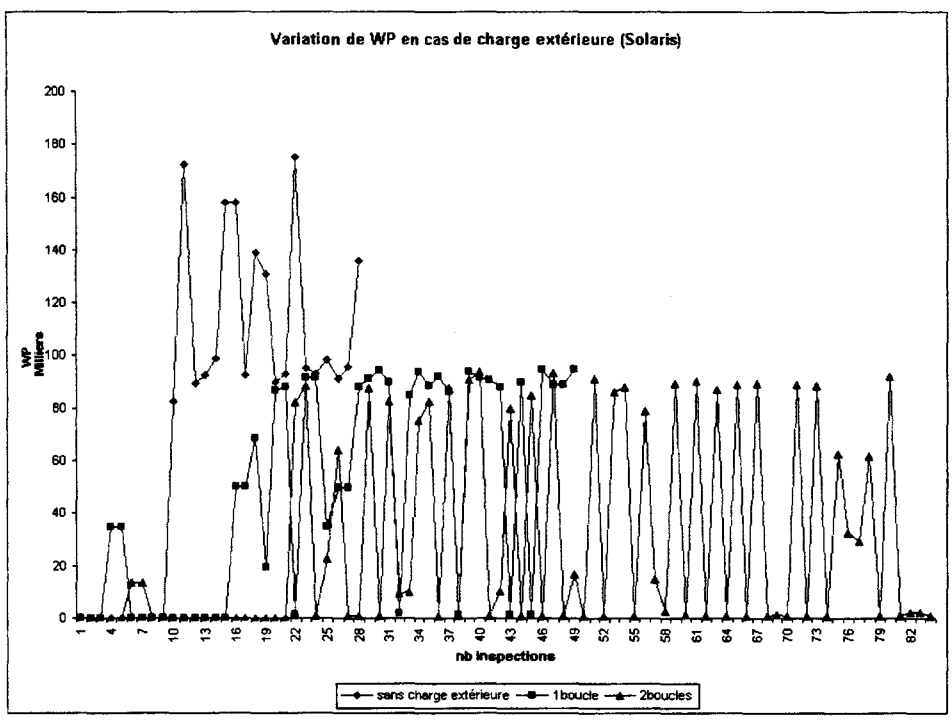




Figure 15.16 – Variation de la quantité de travail en présence de la charge extérieure (Linux et Solaris)

est réalisée avec un seuil petit, mais qui risque de générer d'instabilité par la suite. Une correction plus lente est acquise avec un seuil plus grand, qui accepte des déséquilibres plus importants entre les valeurs de charge. Par défaut, ADAJ implémente l'algorithme algo50 de détection d'un déséquilibre, avec un seuil de 0,3. L'intérêt serait de pouvoir adapter le seuil pendant l'exécution de l'application.

Le rôle des communications intervient dans la politique de localisation. Des objets sont migrés en fonction de la charge de la machine destinataire et des communications existantes avec les objets accueillis sur cette machine. Des décisions "aveugles" de placement peuvent améliorer la distribution de la charge, mais n'ont pas un effet positif sur les communications.

# Quatrième partie

# Conclusions et perspectives

Dans ce travail, nous avons étudié les techniques pour concevoir un environnement de programmation et d'exécution d'applications Java. Dans le cas d'un environnement d'exécution formé par des réseaux de stations de travail, se pose le problème de l'hétérogénéité. Un environnement de développement d'applications est censé cacher l'architecture multiprocesseur en offrant une transparence d'accès aux ressources disponibles. Rendre la conception plus aisée et indépendante des caractéristiques du support d'exécution devient un enjeu majeur. Généralement, le gain en expression est perdu en efficacité : des outils, complexes, de gestion des communications ou de synchronisation sont prévus, qui génèrent un certain surcoût à l'exécution. L'intérêt serait d'assurer une exécution efficace des traitements, par un déploiement automatique ou quasi-automatique des applications, qui profiterait au maximum de la disponibilité des ressources.

Les principales contributions de ce travail résident aux niveaux de la proposition des outils conceptuels pour l'expression du parallélisme, et de la conception d'un mécanisme d'équilibrage de charge dynamique et transparent, au niveau de middleware. Ces propositions ont permis la réalisation du projet ADAJ, Applications Distribuées Adaptatives en Java, environnement de développement et d'exécution d'applications Java parallèles et distribuées. Ces contributions et les perspectives de ce travail sont décrites dans les sections suivantes.

## Contributions

Nous avons présenté dans un premier temps l'environnement de programmation en ADAJ et les apports sur la méthodologie de programmation distribuée et parallèle.

Les *collections distribuées* corrélées avec les *appels asynchrones* induisent en ADAJ une méthodologie de programmation parallèle de type MIMD. ADAJ permet ainsi :

- d'exprimer aisément un parallélisme d'objets dans lequel des traitements peuvent être activés de manière parallèle et asynchrone et qui autorise surtout une récupération totalement asynchrone des résultats,
- d'exprimer un parallélisme de méthodes,
- d'accroître naturellement la granularité des traitements en activant des méthodes sur les groupes d'objet et non sur les objets individuels ou globalement dispersés,
- de ne pas nécessairement fixer lors de la conception du programme parallèle la granularité et le degré du parallélisme, mais de repousser ces choix au moment du lancement effectif du programme, en fonction du jeu réel des données à traiter et des caractéristiques du support d'exécution (en particulier le nombre de machines disponibles). Le degré de parallélisme peut même varier dynamiquement pendant l'exécution.
- d'être libéré des contraintes de localisation explicite des objets et de leurs migrations éventuelles.

Nos objectifs ne sont pas de faire de l'extraction automatique du parallélisme, mais de proposer un modèle de programmation parallèle, apportant un maximum de transparence, à la fois pour simplifier le travail du programmeur (en cachant les problèmes complexes dus à l'écriture d'un programme parallèle) et pour améliorer l'efficacité d'exécution des applications. Le concept de collection distribuée qu'ADAJ propose [FTD01, FDLT01] marie les deux modèles de programmation parallèle : le parallélisme d'objets et le parallélisme de méthodes.

Les *fragments*, éléments d'une collection distribuée, constitue la granularité du parallélisme, gérée explicitement par le programmeur. ADAJ associe, aux collections distribuées, des opérateurs permettant l'activation parallèle et asynchrone de méthodes sur les fragments. ADAJ autorise également l'expression d'un parallélisme de tâches classique, qui provient des appels asynchrones.

Grâce aux techniques de multithreading et d'activation à distance fournies dans le langage

Java, ADAJ offre la transparence d'expression du parallélisme, en cachant la gestion des threads, des exceptions, et gérant les retours des résultats.

A part le niveau conceptuel enrichi, ADAJ s'oriente vers une recherche d'expression proche de Java en termes de syntaxe et en respect de typage. La solution qui passe par la génération de code offre une écriture facile et du typage de l'expression du parallélisme [FT02].

Nous avons montré sur des exemples simples (manipulation des vecteurs, multiplication des matrices), l'approche méthodologique en ADAJ. D'autres exemples plus complexes (d'optimisation combinatoire [FTD03] ou de datamining) ont illustré la nécessité de disposer des outils parallèles au niveau conceptuel des applications.

L'efficacité de l'exécution des programmes distribuées et parallèles est obtenue, en ADAJ, non seulement à travers des outils conceptuels [FTD03], mais aussi grâce au mécanisme d'équilibrage de charge [Fel02, FT03], qui peut dynamiquement adapter l'exécution à l'évolution de l'application. Les applications visées sont de granularité moyenne et irrégulières, aspect caractérisé par des graphes de dépendance entre les objets, sémi-prévisibles ou imprévisibles.

En ADAJ, la recherche d'une exécution efficace des traitements distribués passe par l'exploitation d'un mécanisme d'observation des relations entre les objets d'une application. Ce mécanisme permet d'acquérir une connaissance du comportement de l'application. L'observation de l'application se traduit dans un graphe d'objets, distribué et dynamique, dont les arcs sont étiquetés du nombre d'invocations. Le choix d'observer ce nombre d'invocations a une double explication : une mise en œuvre facile et une approximation correcte de l'activité des objets, aussi bien en termes de travail propre que de communications. L'hypothèse de base est que l'activité d'un objet dans le futur proche est très semblable à son activité dans le passé.

Un deuxième type d'observation disponible en ADAJ surveille la charge du système, information qui aurait pu permettre l'adaptation de l'éxécution également aux caractéristiques de l'environnement d'exécution (hétérogénéité des ressources ou charge extérieure de l'application). L'exploitation de cette information s'est avérée difficile, et en conséquence, le mécanisme d'équilibrage de charge en ADAJ est prévu pour un système dédié, homogène.

Contrairement aux approches d'équilibrage de charge par partitionnement dynamique de graphe, la distribution de charge en ADAJ corrige les mauvaises situations, les déséquilibres, sans avoir pour objectif une solution optimale.

La stratégie d'équilibrage en ADAJ comporte deux composants, de détection d'un déséquilibre et de sa correction. Un critère important est le choix de la politique de détection, qui déclenche la redistribution d'objets. La détection du déséquilibre est basée sur les algorithmes à double seuil, identifiant trois types de valeurs : normales, en dessous des normales et au dessus des normales. A partir de l'étude des méthodes statistiques, les termes d'intervalles de normalité sont définis à base de la moyenne, l'écart moyen et type. Les valeurs intervenant dans cette analyse sont les charges des machines. La charge d'une machine est définie à partir des informations d'observation de relations (mesurant le travail d'une machine virtuelle) et du nombre de threads. La centralisation de ces informations, de manière périodique, permet une décision assez correcte de la charge globale du système, les décisions ultérieures des politiques de sélection et de localisation étant distribuées.

La redistribution d'objets se base sur leur propriété de migration, passive de JavaParty ou active d'ADAJ (par l'extension du mécanisme de JavaParty, en permettant la migration de fragments ayant une méthode active). Une autre technique de déploiement de l'application, le placement initial, est utilisée lors de la création d'objets, pour aider à l'équilibrage de charge.

# Perspectives

L'hétérogénéité caractérise la variété des performances des machines en termes de :
- vitesse du processeur,
- charge du système (dans un environnement multi-utilisateurs),
- type du système.

L'hétérogénéité des machines devrait se refléter dans le mécanisme de distribution pour intégrer un équilibrage inter-applications :
- la définition de la charge ne doit pas uniquement prendre en compte la charge générée par l'application. La charge de la machine, elle-même corrélée avec sa puissance, intervient aussi car un système qui fonctionne deux fois plus rapidement qu'un autre acceptera deux fois plus de travail à exécuter.
- la politique de décision doit prendre en compte les situations d'une charge extérieure à l'application.

Ainsi, l'extension du mécanisme d'équilibrage de charge nécessitera un calibrage des puissances des machines (en termes de vitesse du processeur et/ou de mémoire disponible) et exploitera un mécanisme portable d'évaluation de la charge d'un système multi-utilisateurs.

## Equilibrage de charge inter-applications versus intra-application

La figure 15.17 résume les deux mécanismes d'équilibrage inter et intra-application, mettant en évidence les liaisons entre eux.

Dans un environnement homogène, dans lequel les machines ont la même puissance, potentiellement les quantités de travail à exécuter par les JVM sont égales. Ces quantités potentielles de travail sont appelées *potentiels de la JVM*. Le mécanisme d'équilibrage intra-application est censé fragmenter l'application, en la répartissant sur les machines, pour que les travaux générés soient égaux. Pour un environnement hétérogène et dédié, le potentiel de chaque JVM serait proportionnel à la puissance de la machine.

Dans un contexte hétérogène, non dédié, les CPU ne sont pas disponibles à 100 % pour l'application, parce que les machines partagent, a priori, la CPU entre plusieurs processus de la machine. Dans la figure 15.17, un exemple est donné pour trois types de machines : normale, rapide et lente en termes de vitesse d'exécution. Les potentiels de la JVM varient en fonction de la charge créée par d'autres processus. L'équilibrage de charge inter-applications cherche le déploiement de l'application, sur les machines, qui profite au maximum des potentiels des JVM.

L'information sur la charge de la JVM est disponible en ADAJ à travers des mesures d'observation. En Charm++ [BK99], l'indicateur de charge est la disponibilité du processeur, mesurée en temps CPU obtenu par le processus pendant une période de temps, divisé par le temps écoulé. En ADAJ, si le potentiel de la JVM était connu, la charge d'une machine physique pourrait être estimée par :

$$charge_{mach\ phys} = \frac{charge\ JVM}{potentiel\ JVM} ,$$

qui correspond à la correction de la mesure de charge de la JVM en fonction de la charge extérieure de la machine (par exemple, si une JVM dispose que de la moitié du temps CPU, donc son potentiel est à 50 %, alors la charge de la machine est le double de la charge de la JVM).

De manière incrémentale, cette mesure de charge devra encore être corrigée avec un indice de performance de la machine, $IP$ (indice qui peut être déterminé par une phase de calibrage) :
- $IP < 1$ si la machine est lente,
- $IP = 1$ si la machine est normale,

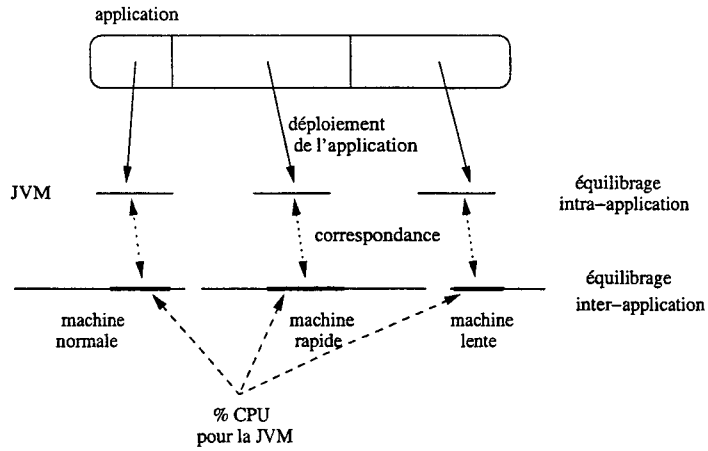



Figure 15.17 – Liaison entre les équilibrages inter et intra application

– $IP > 1$ si la machine est rapide.

La nouvelle mesure sera donc $charge_{corrigee} = \frac{charge\ JVM}{potentiel\ JVM * IP}$. Le mécanisme de charge intra-application pourrait être appliqué en prenant comme valeurs de charge ces nouvelles mesures, qui tiennent compte à la fois de la charge extérieure de l'application et de la performance de la machine physique.

L'approche sera orientée vers une correction d'une valeur de charge intra-application disponible, pour en déduire une valeur de charge inter-applications.

## Paramétrage

Un autre aspect qui mérite une réflexion importante est le paramétrage de l'algorithme d'équilibrage de charge : quelles sont les meilleures valeurs pour la fréquence d'inspection, de lissage des mesures d'observation ? Comment les seuils seront-ils fixés (au départ, et éventuellement modifiés en fonction de l'évolution de l'application) ? Une question se pose : y a-t-il une dépendance entre ces paramètres ?

# Cinquième partie

# Annexes

# Annexe A

# Génération du code pour les appels asynchrones

La génération de code pour les appels asynchrones est réalisée de manière similaire à la génération de code pour les appels du type distribute. La figure suivante montre le procédé.

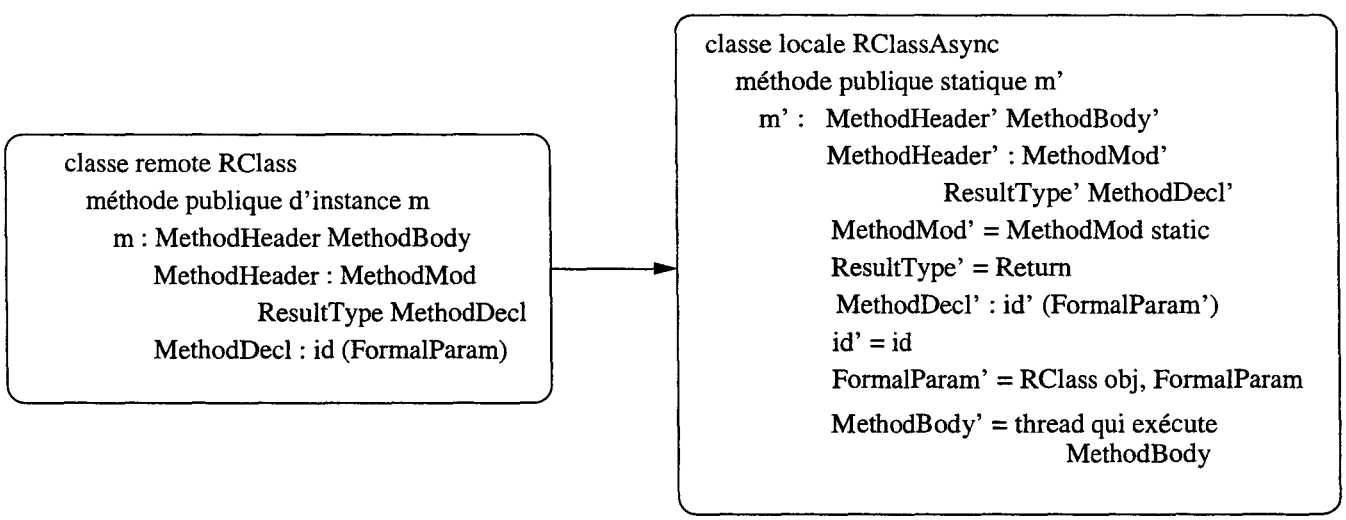



Figure A.1 – Génération des classes pour les appels asynchrones

# Annexe B

# Méthodologie de programmation

Le code ADAJ suivant correspond à l'algorithme distribué et parallèle de la recherche de la plus longue sous-suite croissante d'une série d'entiers, version sans communications entre les sous-séries.

```
package suitef;

import java.util.Vector;
import fr.lifl.adaj.parallel.RemoteFragment;
import jp.lang.DistributedRuntime;

public class SousSuite extends RemoteFragment{
  public int longueur;
  Vector sousSuite;
  Object[] res = new Integer[5];
  // contenu de res:
  //   - premier element
  //   - longueur de la premiere sous-suite
  //   - longeur maximale
  //   - dernier element
  //   - longueur de la derniere sous-suite

  public SousSuite() {
      super();
  }

  public void setSousSuite( Object[] sousSuite) {
    int indGChaine = sousSuite.length;
    this.sousSuite = new Vector (indGChaine);
    for (int i = 0; i < indGChaine; i++)
      this.sousSuite.add(sousSuite[i]);
      this.longueur = this.sousSuite.size();
      System.out.println("vector " + this.sousSuite);
    }

  public int getLong() {
    return sousSuite.size();
```

```
}

public void premiereSuite() {
  int index = 0;
  int longSuite = 1;
  boolean suite = true;
  while (suite && index < longueur - 1){
    if (((Integer)sousSuite.elementAt(index)).compareTo(
                  sousSuite.elementAt(index+1)) >0 )
       suite = false;
    else{
       longSuite++;
       index++;
    }
  }
  res[0] = sousSuite.elementAt(0);
  res[1] = new Integer(longSuite);
}

public void derniereSuite(){
  int index = longueur - 1;
  int longsuite = 1;
  boolean suite = true;
  while (suite && index > 0){
    if (((Integer)sousSuite.elementAt(index)).compareTo(
                  sousSuite.elementAt(index-1)) < 0)
       suite = false;
    else{
       longsuite++;
       index--;
    }
  }
  res[3] = sousSuite.lastElement();
  res[4] = new Integer(longsuite);
}

public void plusGrandeSuiteMaximale(){
  int index = 0; int indexMax = 0;
  int longsuite = 1;
  int longsuiteMax = 1;
  while (index < longueur - 1){
    while ((index  < longueur - 1)&&
          (((Integer)sousSuite.elementAt(index)).compareTo(
                     sousSuite.elementAt(index + 1)) <= 0)) {
       longsuite++;
       index++;
    }
```

```
      if (longsuite > longsuiteMax){
         longsuiteMax = longsuite;
         indexMax = index - longsuiteMax + 1;
      }
      longsuite = 1;
      index++;
    }
    res[2] = new Integer(longsuiteMax);
  }

  public Object[] calculLocal(){
    premiereSuite();
    derniereSuite();
    plusGrandeSuiteMaximale();
    return res;
  }
}
```

```
//  Sous-suite maximale
// implementation ADAJ avec les collections distribuees
// la construction de la collection distribuee ne se fait pas en parallele
// premiere solution : sans communication entre les fragments

package suitef;

import java.util.Vector;
import fr.lifl.adaj.parallel.DistributedCollection;
import fr.lifl.adaj.parallel.DistributedTask;
import fr.lifl.adaj.parallel.Collector;
import fr.lifl.adaj.parallel.PackRes;
import jp.lang.DistributedRuntime;

public class SuiteMain{
  public static void main(String[] args){
    if (args.length == 1){
       int nmachines = DistributedRuntime.getMachineCnt();
       DistributedCollection cd = new DistributedCollection("suitef.SousSuite");
       for (int i = 0; i < nmachines; i++)
         cd.addFragment();

         // pour des suites issues d'un fichier donne (découpées
         // aléatoirement ou avec des tailles fixes)
         Vector suitesD = mandata.DivideSuiteEqualArray.divide(args[0], nmachines);
         for (int i=0; i < cd.size(); i++)
           // de fichier entier
           ((SousSuite)cd.get(i)).setSousSuite((Object[])suitesD.get(i));

         Collector cRes = DistributedTask.distribute(cd,"calculLocal",null);
```

```
        PackRes res = cRes.getI(0);
        Object[] collector0 = (Object[])res.getResult();
        int maxi = ((Integer)collector0[2]).intValue();
        for (int j = 1; j< cd.size(); j++){
          PackRes resi = cRes.getI(j);
          Object[] collectori = (Object[])resi.getResult();
          if (((Integer)collector0[3]).compareTo(collectori[0]) <= 0){
             if (((Integer)collector0[4]).intValue() ==
                 ((SousSuite)resi.getFragment()).getLong())
                collectori[1] = new Integer(((Integer)collectori[1]).intValue() +
                                            ((Integer)collector0[1]).intValue());
             else
                collectori[1] = new Integer(((Integer)collectori[1]).intValue() +
                                            ((Integer)collector0[4]).intValue());
             if (((Integer)collectori[2]).compareTo(collectori[1]) < 0)
                      collectori[2] = collectori[1];
          }
          if (maxi < ((Integer)collectori[2]).intValue())
             maxi = ((Integer)collectori[2]).intValue();
             collector0 = collectori;
        }
        System.out.println("Longueur de la suite max : " + maxi);
      }
      else
        System.out.println("Usage : jadaj -host ... suitef.SuiteMain f-donnees");
    }
}
```

# Annexe C

# Surcoût brut d'une communication ADAJ

Les tableaux suivants contiennent les temps d'exécution pour le calcul des surcoûts générés uniquement par les communications dans l'utilisation des collections distribuées.

Les programmes testés, sur des plates-formes homogènes et hétérogènes, consistent dans l'invocation :

- d'une méthode de corps vide qui ne retourne rien (*Vide*),
- d'une méthode qui retourne un objet du type `Integer` (*Result*),
- d'une méthode qui exécute une boucle finie et retourne un objet du type `Integer` (*RB*).

### Surcoût communication ADAJ sur un réseau homogène

La plate-forme testée est formée par deux machines de processeur PIII, fréquence 733 MHz et mémoire RAM 128 Mo, ayant comme système d'exploitation Linux 2.4.17.

| Vide 10 | adaj (ms) | jp | Res 10 | adaj (ms) | jp | RB 10 | adaj (ms) | jp |
|---|---|---|---|---|---|---|---|---|
| | 9,7 | 1,8 | | 16 | 2,7 | | 28,9 | 28 |
| | 8,9 | 1,7 | | 15,9 | 2,5 | | 27,7 | 27,1 |
| | 9,4 | 1,7 | | 16,9 | 2,6 | | 28,7 | 27,3 |
| | 8,8 | 1,8 | | 16,2 | 2,6 | | 28,8 | 27,2 |
| | 8,8 | 1,7 | | 14,7 | 2,4 | | 27,1 | 27,2 |
| 439,29 % | 9,3 | 1,6 | 489,31 % | 14,8 | 2,7 | 2,87 % | 26,9 | 27,1 |
| | 8,6 | 1,7 | | 14,6 | 2,5 | | 27,6 | 27,1 |
| | 8,9 | 1,6 | | 15,8 | 2,8 | | 28,7 | 27,6 |
| | 9,4 | 1,6 | | 14,6 | 2,8 | | 27,2 | 27,2 |
| moy | 9,06 | 1,68 | | 15,44 | 2,62 | | 27,98 | 27,2 |

| Vide 100 | adaj (ms) | jp | Res 100 | adaj (ms) | jp | RB 100 | adaj (ms) | jp |
|---|---|---|---|---|---|---|---|---|
| | 6,87 | 1,57 | | 10,2 | 2,63 | | 22,76 | 27,36 |
| | 6,51 | 1,46 | | 9,39 | 2,54 | | 21,61 | 27,25 |
| | 6,44 | 1,52 | | 9,23 | 2,53 | | 21,41 | 27,2 |
| | 6,26 | 1,52 | | 8,99 | 2,45 | | 21,87 | 27,13 |
| | 6,28 | 1,44 | | 9,47 | 2,45 | | 21,35 | 27,28 |
| 323,58 % | 6,22 | 1,56 | 263,81 % | 8,76 | 2,80 | -21,3 % | 21,5 | 27,29 |
| | 6,15 | 1,47 | | 8,96 | 2,45 | | 21,15 | 27,13 |
| | 6,09 | 1,45 | | 8,75 | 2,49 | | 20,97 | 27,12 |
| | 6,23 | 1,38 | | 8,54 | 2,39 | | 20,93 | 27,08 |
| moy | 6,28 | 1,48 | | 9,07 | 2,49 | | 21,40 | 27,2 |

Tableau C.1 – Surcoûts bruts (en %) et temps (en ms) d'une communication ADAJ sur un réseau homogène (10 et 100 appels successifs)

| Vide 1000 | adaj jp (ms) | | Res 1000 | adaj jp (ms) | | RB 1000 | adaj jp (ms) | |
|---|---|---|---|---|---|---|---|---|
| | 4,81 | 1,56 | | 6,8 | 2,36 | | 19,22 | 27 |
| | 4,38 | 1,56 | | 6,17 | 2,27 | | 18,56 | 26,96 |
| | 4,23 | 1,39 | | 6,07 | 2,06 | | 18,54 | 26,69 |
| | 4,24 | 1,40 | | 6,03 | 2,04 | | 18,49 | 26,74 |
| | 4,2 | 1,38 | | 6,04 | 2,06 | | 18,51 | 26,66 |
| 205,01 % | 4,21 | 1,40 | 196,59 % | 6,07 | 2,04 | -30,8 % | 18,57 | 26,75 |
| | 4,22 | 1,39 | | 6,17 | 2,05 | | 18,45 | 26,63 |
| | 4,22 | 1,37 | | 6,05 | 2,04 | | 18,48 | 27,07 |
| | 4,5 | 1,40 | | 6,1 | 2,06 | | 18,5 | 26,67 |
| moy | 4,26 | 1,40 | | 6,09 | 2,05 | | 18,52 | 26,76 |

| Vide 10000 | adaj jp (ms) | | Res 10000 | adaj jp (ms) | | RB 10000 | adaj jp (ms) | |
|---|---|---|---|---|---|---|---|---|
| | 3,34 | 1,20 | | 5,09 | 1,63 | | 17,54 | 26,34 |
| | 3,38 | 1,15 | | 5,01 | 1,58 | | 17,46 | 26,27 |
| | 3,37 | 1,14 | | 5,02 | 1,59 | | 17,47 | 26,28 |
| | 3,36 | 1,11 | | 5,02 | 1,57 | | 17,51 | 26,33 |
| | 3,37 | 1,13 | | 5,04 | 1,59 | | 17,45 | 26,39 |
| 196,13 % | 3,36 | 1,13 | 218,48 % | 4,99 | 1,58 | -33,57 % | 17,48 | 26,3 |
| | 3,37 | 1,14 | | 5,06 | 1,58 | | 17,46 | 26,26 |
| | 3,35 | 1,14 | | 5,04 | 1,57 | | 17,44 | 26,27 |
| | 3,36 | 1,11 | | 5,04 | 1,57 | | 17,45 | 26,26 |
| moy | 3,36 | 1,14 | | 5,03 | 1,58 | | 17,46 | 26,29 |

Tableau C.2 – Surcoûts bruts (en %) et temps (en ms) d'une communication ADAJ sur un réseau homogène (1000 et 10000 appels successifs)

### Surcoût communication ADAJ sur un réseau hétérogène en puissance des machines

La plate-forme testée est formée par deux machines, une de processeur PIII 600 MHz, 128 Mo RAM et système d'exploitation Linux 2.2.17 et l'autre de processeur PIV 1,66 GHz et 256 Mo RAM et système d'exploitation Linux 2.4.2-2.

| Vide 10 | adaj jp (ms) | | Res 10 | adaj jp (ms) | | RB 10 | adaj jp (ms) | |
|---|---|---|---|---|---|---|---|---|
| | 71 | 85 | | 89 | 19 | | 71 | 66 |
| | 80 | 67 | | 87 | 39 | | 107 | 100 |
| | 80 | 80 | | 86 | 77 | | 114 | 100 |
| | 99 | 80 | | 88 | 78 | | 103 | 95 |
| | 92 | 79 | | 156 | 79 | | 109 | 98 |
| 3,01 % | 95 | 80 | 11,53 % | 82 | 78 | 7,87 % | 108 | 118 |
| | 78 | 78 | | 85 | 80 | | 107 | 99 |
| | 77 | 80 | | 87 | 78 | | 108 | 100 |
| | 81 | 80 | | 87 | 80 | | 104 | 98 |
| moy | 82,2 | 79,8 | | 87 | 78 | | 106,8 | 99 |

| Vide 100 | adaj jp (ms) | | Res 100 | adaj jp (ms) | | RB 100 | adaj jp (ms) | |
|---|---|---|---|---|---|---|---|---|
| | 65 | 64 | | 76 | 65 | | 85 | 90 |
| | 75 | 72 | | 78 | 72 | | 86 | 91 |
| | 75 | 74 | | 76 | 73 | | 87 | 90 |
| | 74 | 70 | | 91 | 72 | | 85 | 89 |
| | 75 | 74 | | 151 | 73 | | 86 | 90 |
| 2,72 % | 90 | 76 | 6,06 % | 76 | 73 | -4,44 % | 86 | 91 |
| | 101 | 114 | | 77 | 72 | | 90 | 88 |
| | 74 | 74 | | 76 | 72 | | 87 | 89 |
| | 79 | 74 | | 78 | 72 | | 85 | 91 |
| moy | 75,6 | 79,8 | | 77 | 73 | | 86 | 90 |

Tableau C.3 – Surcoûts bruts (en %) et temps (en ms) d'une communication ADAJ sur un réseau hétérogène (10 et 100 appels successifs)

| Vide 1000 | adaj (ms) | jp | Res 1000 | adaj (ms) | jp | RB 1000 | adaj (ms) | jp |
|---|---|---|---|---|---|---|---|---|
| | 73 | 78 | | 75 | 72 | | 82 | 89 |
| | 72 | 74 | | 74 | 72 | | 80 | 88 |
| | 74 | 71 | | 78 | 78 | | 82 | 88 |
| | 77 | 86 | | 76 | 72 | | 130 | 88 |
| | 75 | 75 | | 74 | 72 | | 93 | 89 |
| -4,92 % | 73 | 80 | 3,32 % | 74 | 78 | 0,68 % | 91 | 88 |
| | 74 | 92 | | 75 | 73 | | 93 | 87 |
| | 73 | 71 | | 75 | 72 | | 99 | 87 |
| | 72 | 79 | | 74 | 72 | | 85 | 99 |
| moy | 73,4 | 77,2 | | 74,6 | 72 | | 88,8 | 88,2 |

Tableau C.4 – Surcoûts bruts (en %) et temps (en ms) d'une communication ADAJ sur un réseau hétérogène (1000 appels successifs)

## Surcoût communication ADAJ sur un réseau hétérogène en système d'exploitation

La plate-forme testée est formée par deux machines, une de processeur 600 MHz, 128 Mo RAM et système d'exploitation Linux 2.2.17 et l'autre un sparc ultra5-10, de système d'exploitation SunOS 5.7.

| Vide 10 | adaj (ms) | jp | Res 10 | adaj (ms) | jp | RB 10 | adaj (ms) | jp |
|---|---|---|---|---|---|---|---|---|
| | 27 | 77 | | 96 | 23 | | 79 | 59 |
| | 72 | 79 | | 90 | 62 | | 118 | 106 |
| | 81 | 79 | | 88 | 78 | | 114 | 117 |
| | 78 | 80 | | 90 | 79 | | 124 | 119 |
| | 84 | 77 | | 92 | 77 | | 122 | 119 |
| 0,52 % | 80 | 76 | 14,57 % | 91 | 81 | -2,04 % | 114 | 120 |
| | 78 | 80 | | 86 | 79 | | 115 | 118 |
| | 79 | 76 | | 89 | 80 | | 103 | 121 |
| | 75 | 73 | | 88 | 78 | | 116 | 116 |
| moy | 78 | 77,6 | | 89,6 | 78 | | 115,4 | 118 |

| Vide 100 | adaj (ms) | jp | Res 100 | adaj (ms) | jp | RB 100 | adaj (ms) | jp |
|---|---|---|---|---|---|---|---|---|
| | 73 | 65 | | 72 | 65 | | 90 | 103 |
| | 74 | 72 | | 78 | 73 | | 95 | 111 |
| | 74 | 72 | | 80 | 73 | | 94 | 111 |
| | 75 | 71 | | 77 | 72 | | 94 | 110 |
| | 73 | 73 | | 78 | 72 | | 94 | 111 |
| 1,95 % | 72 | 72 | 7,97 % | 80 | 73 | -15,01 % | 94 | 111 |
| | 72 | 71 | | 83 | 73 | | 97 | 111 |
| | 72 | 72 | | 77 | 73 | | 97 | 111 |
| | 73 | 73 | | 80 | 73 | | 94 | 112 |
| moy | 73,2 | 71,8 | | 78,6 | 73 | | 94,2 | 111 |

Tableau C.5 – Surcoûts bruts (en %) et temps (en ms) d'une communication ADAJ sur un réseau hétérogène (10 et 100 appels successifs)

| Vide 1000 | adaj (ms) | jp | Res 1000 | adaj (ms) | jp | RB 1000 | adaj (ms) | jp |
|---|---|---|---|---|---|---|---|---|
| | 77 | 71 | | 74 | 71 | | 90 | 110 |
| | 72 | 72 | | 76 | 71 | | 91 | 110 |
| | 72 | 72 | | 75 | 72 | | 92 | 110 |
| | 71 | 72 | | 75 | 72 | | 91 | 110 |
| | 71 | 72 | | 75 | 72 | | 91 | 110 |
| -0,56 % | 71 | 72 | 4,17 % | 72 | 20,53 | -17,3 % | 91 | 110 |
| | 72 | 71 | | 75 | 72 | | 91 | 111 |
| | 72 | 72 | | 76 | 72 | | 91 | 110 |
| | 72 | 72 | | 75 | 72 | | 91 | 110 |
| moy | 71,6 | 72 | | 75 | 72 | | 91 | 110 |

Tableau C.6 – Surcoûts bruts (en %) et temps (en ms) d'une communication ADAJ sur un réseau hétérogène (1000 appels successifs)

# Annexe D

# Coût de migration JavaParty

Les différents cas testés pour le calcul du coût de migration JavaParty sont :
- cas 1 : objet vide,
- cas 2 : objet vecteur, à 100 objets nuls,
- cas 3 : objet vecteur, à 100 objets du type `Integer`,
- cas 4 : objet contenant 100 vecteurs, à 100 objets du type `Integer`.

## D.1 Coût de migration sur un réseau homogène

L'environnement homogène est constitué de deux machines, à processeur PIII à 733 MHz et 128 Mo de capacité mémoire, sous la plate-forme Linux 2.2.17 (Debian 2.2, liées par un réseau de débit 100 Mb/s. Les coûts obtenus sont montrés dans les tableaux D.1 et D.2.

## D.2 Coût de migration sur un réseau hétérogène

L'environnement hétérogène est constitué de deux machines, une à processeur PIII à 600 MHz, 128 Mo RAM, sous Linux 2.2.17 avec la version jdk1.3.0rc1 Java et l'autre, un processeur PIV à 1,66 GHz, 256 Mo RAM, sous Linux 2.2.2_2, avec la version 1.3.1 de Java), liées par un réseau de débit 10 Mb/s. Les coûts obtenus sont montrés dans les tableaux D.3 et D.4.

| cas 1 | n=10 | n=20 | n=30 | n=40 |
|---|---|---|---|---|
| | 22 | 21 | 20 | 20 |
| | 20 | 18 | 15 | 16 |
| | 15 | 15 | 15 | 15 |
| | 15 | 16 | 15 | 14 |
| | 16 | 14 | 15 | 14 |
| | 14 | 15 | 14 | 13 |
| | 14 | 14 | 13 | 13 |
| | 15 | 14 | 13 | 13 |
| | 15 | 14 | 13 | 13 |
| moy | 15,2 | 14,8 | 14,4 | 13,8 |

| cas 2 | n=10 | n=20 | n=30 | n=40 |
|---|---|---|---|---|
| | 30 | 28 | 28 | 26 |
| | 21 | 19 | 18 | 17 |
| | 19 | 16 | 17 | 15 |
| | 15 | 18 | 16 | 15 |
| | 17 | 18 | 15 | 15 |
| | 17 | 16 | 15 | 14 |
| | 17 | 15 | 15 | 14 |
| | 16 | 14 | 14 | 14 |
| | 16 | 15 | 14 | 14 |
| moy | 17,2 | 16,6 | 15,6 | 14,6 |

Tableau D.1 – Coûts (en ms) de la migration JavaParty sous en réseau homogène

| cas 3 | n=10 | n=20 | n=30 | n=40 |
|---|---|---|---|---|
| | 43 | 37 | 32 | 29 |
| | 24 | 20 | 19 | 18 |
| | 21 | 18 | 19 | 17 |
| | 18 | 19 | 18 | 17 |
| | 17 | 18 | 17 | 17 |
| | 18 | 18 | 17 | 16 |
| | 19 | 17 | 18 | 16 |
| | 18 | 17 | 17 | 16 |
| | 18 | 17 | 16 | 16 |
| moy | 18,8 | 18,0 | 17,8 | 16,6 |

| cas 4 | n=10 | n=20 | n=30 | n=40 |
|---|---|---|---|---|
| | 329 | 307 | 301 | 300 |
| | 301 | 295 | 274 | 287 |
| | 305 | 270 | 280 | 283 |
| | 305 | 299 | 289 | 274 |
| | 275 | 272 | 278 | 284 |
| | 323 | 269 | 267 | 275 |
| | 277 | 274 | 292 | 276 |
| | 275 | 280 | 267 | 273 |
| | 285 | 257 | 302 | 286 |
| moy | 294,6 | 278,2 | 282,6 | 280,8 |

Tableau D.2 – Coûts (en ms) de la migration JavaParty sous en réseau homogène

| cas 1 | n=10 | n=20 | n=30 | n=40 | n=50 |
|---|---|---|---|---|---|
| | 45 | 30 | 34 | 30 | 27 |
| | 30 | 33 | 24 | 24 | 23 |
| | 48 | 24 | 23 | 28 | 66 |
| | 36 | 32 | 23 | 22 | 21 |
| | 24 | 22 | 22 | 21 | 21 |
| | 33 | 22 | 22 | 23 | 21 |
| | 24 | 22 | 22 | 21 | 20 |
| | 23 | 30 | 22 | 23 | 22 |
| | 24 | 23 | 21 | 21 | 20 |
| moy | 29,4 | 25,8 | 22,4 | 22,6 | 21,6 |

| cas 2 | n=10 | n=20 | n=30 | n=40 | n=50 |
|---|---|---|---|---|---|
| | 43 | 32 | 31 | 29 | 42 |
| | 31 | 27 | 28 | 24 | 24 |
| | 30 | 25 | 24 | 23 | 23 |
| | 27 | 25 | 29 | 23 | 24 |
| | 27 | 24 | 23 | 23 | 22 |
| | 65 | 23 | 25 | 22 | 22 |
| | 25 | 25 | 25 | 23 | 26 |
| | 28 | 22 | 22 | 23 | 21 |
| | 26 | 23 | 22 | 21 | 21 |
| | 24 | 25 | | | |
| moy | 28,17 | 24,5 | 25,0 | 23,0 | 23,0 |

Tableau D.3 – Coûts (en ms) de la migration JavaParty sous en réseau hétérogène

| cas 3 | n=10 | n=20 | n=30 | n=40 | n=50 |
|---|---|---|---|---|---|
| | 47 | 40 | 30 | 27 | 26 |
| | 36 | 30 | 26 | 26 | 26 |
| | 29 | 26 | 29 | 28 | 27 |
| | 27 | 26 | 25 | 25 | 27 |
| | 30 | 26 | 51 | 25 | 26 |
| | 27 | 26 | 25 | 27 | 27 |
| | 27 | 26 | 24 | 24 | 34 |
| | 27 | 39 | 26 | 25 | 25 |
| | 27 | 25 | 26 | 28 | 27 |
| | 27 | 24 | 25 | 25 | 25 |
| moy | 27,83 | 26,67 | 26,17 | 25,83 | 26,5 |

| cas 4 | n=10 | n=20 | n=30 | n=40 | n=50 |
|---|---|---|---|---|---|
| | 308 | 292 | 299 | 288 | 295 |
| | 415 | 296 | 281 | 284 | 278 |
| | 308 | 292 | 286 | 280 | 276 |
| | 307 | 291 | 293 | 270 | 278 |
| | 283 | 288 | 358 | 277 | 284 |
| | 280 | 284 | 315 | 280 | 278 |
| | 271 | 295 | 306 | 280 | 271 |
| | 293 | 278 | 288 | 271 | 277 |
| | 289 | 285 | 542 | 294 | 278 |
| | 716 | 313 | 315 | | 275 |
| moy | 298,0 | 290,5 | 302,67 | 280,2 | 277,5 |

Tableau D.4 – Coûts (en ms) de la migration JavaParty sous en réseau hétérogène

# Annexe E

# Inférence de types

## E.1 Signatures

Le bytecode contient beaucoup de types écrits sous forme de signatures, notamment pour les appels de méthodes. La signature d'une méthode a la forme générale :

(types des paramètres)type du résultat

- les types de base sont codés par une lettre,
- les objets sont codés par le nom de la classe encadré par un $L$ et un ;,
- les vecteurs sont codés en mettant devant autant de "[" qu'il y a de dimensions.

Les codes utilisés sont :

| | | | | | |
|---|---|---|---|---|---|
| V | void | D | double | J | long |
| B | byte | F | float | S | short |
| C | char | I | int | Z | boolean |

Par exemple, $(IDLjava/lang/String;)Ljava/lang/String;$ désigne la signature d'une méthode qui prend comme paramètres un entier, un double et une chaîne de caractères et renvoie une chaîne de caractères.

## E.2 Algorithme d'inférence de types

Pour chaque instruction de la méthode, on associe : un booléen, une pile d'opérandes et un vecteur de variables locales, mémorisant l'état avant l'exécution "simulée" de cette instruction. Ce qui nous intéresse, ici, ce sont les types des objets et des types primitifs manipulés. La pile et les variables peuvent contenir

- soit un nom complet de classe pour les objets (par exemple, "java.lang.String"), soit un code pour les types primitifs (par exemple, INT, FLOAT, LONG, DOUBLE), la machine virtuelle ne faisant pas de distinction entre les types : entiers, char, byte, ou short ; et les booléens sont traités comme des entiers,
- soit une référence nulle pour les variables non initialisées.

Ensuite un analyseur de flot de données est initialisé. Pour la première instruction de la méthode, les variables contiennent des valeurs déduites des types des paramètres, nulle sinon ; le booléen est vrai ; la pile est vide.

L'analyseur de flot de données est alors lancé. Pour chaque instruction, le booléen indique si l'instruction doit être examinée (initialement le booléen n'est vrai que pour la première instruction). L'analyseur effectue la boucle suivante :

Etape 1 : Sélectionner une instruction dont le booléen est vrai. Si aucune instruction n'a son booléen à vrai, l'algorithme est terminé. Sinon, mettre le booléen, de l'instruction suivante, à faux.

Etape 2 : Simuler l'effet de l'instruction sur la pile et les variables, sous forme de manipulation de types. La plupart des instructions se contentent de dépiler et d'empiler des types. Un petit nombre d'instructions nécessitent une simulation plus soignée : l'accès aux attributs, l'accès aux variables, vecteurs par exemple.

Etape 3 : Déterminer les instructions qui peuvent suivre l'instruction courante. Ces instructions suivantes peuvent être :

- L'instruction suivante, si l'instruction courante n'est pas un branchement systématique (par exemple, `goto`, `return`, `athrow`).
- Les cibles des instructions conditionnelles ou non conditionnelles et des "switch"s
- Les gestionnaires d'exceptions de cette instruction.

Etape 4 Fusionner l'état de la pile et des variables à la fin de l'exécution de l'instruction courante avec la pile et les variables de chaque instruction suivante. Dans le cas d'un gestionnaire d'exception, le pile est réduite à un seul élément dont la valeur est déduite du type de l'exception.

- Si l'instruction suivante est visitée pour la première fois, il n'y a pas de fusion, mais copie de la pile et des variables calculées à l'étape 2. Le booléen de l'instruction suivante est mis à vrai.
- Si l'instruction suivante a été déjà visitée, fusionner la pile et les variables calculées dans l'étape 2 avec la pile et les variables existant dans l'instruction suivante. Le booléen de l'instruction suivante est mis à vrai, s'il y eut des modifications lors de la fusion.

## E.3 La fusion des piles et des variables

Pour fusionner deux piles, les tailles des piles doivent être les mêmes. Pour tous les couples d'éléments correspondants dans les deux piles, les types de ces éléments doivent être les mêmes, sauf pour les références d'objets, auquel cas, la première super-classe, commune aux deux types, est mise dans la pile.

Il en est de même, pour fusionner deux vecteurs de variables. Ils ont toujours la même taille ; il faut que pour tous les couples d'éléments correspondants dans les deux vecteurs, les types de ces éléments soient les mêmes, sinon un type indéfini est mis dans la variable, sauf pour les références d'objets, auquel cas, la première super-classe, commune aux deux types, est mise dans la variable.

## E.4 La sauvegarde de la pile

Pile au début
abbc
Construction du paquet

| | |
|---|---|
| new <PaquetPile> | a b b c p |
| dup | a b b c p p |
| push 3 | a b b c p p 3 |
| invokespecial PaquetPile.<init>(I)V | a b b c p |

Traitement des éléments, en fonction de leur type :
- élément objet

```
    dup_x1                                          a b b p c p
    swap                                            a b b p p c
    invokespecial PaquetPile.addVar(LObject;)V      a b b p
- élément double (long, double)
    dup_x2                                  a p b b p
    dup_x2                                  a p p b b p
    pop                                     a p p b b
    invokespecial PaquetPile.addVar(D)V     a p
- élément simple (boolean, byte, short, int, char, float)
    dup_x1                                  p a p
    swap                                    p p a
    invokespecial PaquetPile.addVar(I)V     p
Mise en forme et rangement d'attribut
 aload_0                            p x
 swap                               x p
 putfield RemoteFragment.pile
```

## E.5 La restauration de la pile

```
Chargement d'attribut
 aload_0                x
 getfield Paquet.pile   p
Restauration des éléments (en fonction de leur type) :
- élément simple (boolean, byte, short, int, char, float)
    dup                                     p p
    invokespecial PaquetPile.intVar()I      p a
    swap                                    a p
- élément double
    dup                                         a p p
    invokespecial PaquetPile.doubleVar()D       a p b b
    dup2_x1                                     a b b p b b
    pop2                                        a b b p
- élément objet
    dup                                             a b b p p
    invokespecial PaquetPile.objectVar()LObject;    a b b p c
    checkcast <String>                              a b b p c
    swap                                            a b b c p
Mise en forme
 pop   a b b c
```

## E.6 Classe paquets variable et pile

### E.6.1 Classe fr.lifl.adaj.PaquetVars

```
public class fr.lifl.adaj.PaquetVars extends java.lang.Object
                                     implements java.io.Serializable {
    protected java.lang.Object objets[] = null; // variables
    protected transient int index = 0;          // indice dans objets
```

```
    public PaquetVars(int n) { // constructeur
      if (n != 0) objects = new Object[n];
    }
    // ajout au paquet
    public void addVar(double i) { objects[index++] = new Double(i); }
    public void addVar(float i)  { objects[index++] = new Float(i); }
    public void addVar(int i)    { objects[index++] = new Integer(i); }
    public void addVar(long i)   { objects[index++] = new Long(i); }
    public void addVar(Object o) { objects[index++] = o; }

    // restauration a partir du paquet
    public double doubleVar() { return ((Double)objects[index++]).doubleValue();}
    public float  floatVar()  { return ((Float)objects[index++]).floatValue();}
    public int    intVar()    { return ((Integer)objects[index++]).intValue();}
    public long   longVar()   { return ((Long)objects[index++]).longValue();}
    public Object objectVar() { return objects[index++];}
}
```

### E.6.2 Classe fr.lifl.adaj.PaquetPile

```
public class fr.lifl.adaj.PaquetPile extends java.lang.Object
                                     implements java.io.Serializable {
    protected java.lang.Object objets[] = null; // variables
    protected transient int index = 0;       // indice dans objets
    public PaquetPile(int n) { // constructeur
      if (n != 0) objects = new Object[n]; index = n;
    }
    // ajout au paquet
    public void addVar(double i) { objects[--index] = new Double(i); }
    public void addVar(float i)  { objects[--index] = new Float(i); }
    public void addVar(int i)    { objects[--index] = new Integer(i); }
    public void addVar(long i)   { objects[--index] = new Long(i); }
    public void addVar(Object o) { objects[--index] = o; }

    // restauration a partir du paquet
    public double doubleVar() { return ((Double)objects[index++]).doubleValue();}
    public float  floatVar()  { return ((Float)objects[index++]).floatValue();}
    public int    intVar()    { return ((Integer)objects[index++]).intValue();}
    public long   longVar()   { return ((Long)objects[index++]).longValue();}
    public Object objectVar() { return objects[index++];}
}
```

## E.7 Exemple de transformation de bytecode

Soit la méthode

```
public void bcl () {
  int i, j;
```

```
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            System.out.print(i + " " + j + " ");
}
```

Le bytecode de la méthode, obtenu grâçe à, l'outil `javap` est le suivant : Method void bcl()
0 iconst_0
1 istore_1
2 goto 55
5 iconst_0
6 istore_2
7 goto 47
10 getstatic #2 <Field java.io.PrintStream out>
13 new #3 <Class java.lang.StringBuffer>
16 dup
17 invokespecial #4 <Method java.lang.StringBuffer()>
20 iload_1
21 invokevirtual #5 <Method java.lang.StringBuffer append(int)>
24 ldc #6 <String " ">
26 invokevirtual #7 <Method java.lang.StringBuffer append(java.lang.String)>
29 iload_2
30 invokevirtual #5 <Method java.lang.StringBuffer append(int)>
33 ldc #6 <String " ">
35 invokevirtual #7 <Method java.lang.StringBuffer append(java.lang.String)>
38 invokevirtual #8 <Method java.lang.String toString()>
41 invokevirtual #9 <Method void print(java.lang.String)>
44 iinc 2 1
47 iload_2
48 iconst_3
49 if_icmplt 10
52 iinc 1 1
55 iload_1
56 iconst_3
57 if_icmplt 5
60 return

Le bytecode transformé est le suivant (les transformations sont marquées en gras) :

Method void bcl()
**0 aload_0**
**1 getfield #190 <Field int etat>**
**4 ifeq 67**
**7 aload_0**
**8 invokevirtual #216 <Method void aMigre()>**
**11 aload_0**
**12 getfield #193 <Field int no>**
**15 tableswitch 0 to 1 : default 67**
**0 : 36**

```
  1 : 49
36 aload_0
37 getfield #205 <Field fr.lifl.adaj.PaquetVars vars>
40 dup
41 invokevirtual #213 <Method int intVar()>
44 istore_1
45 pop
46 goto 72
49 aload_0
50 getfield #205 <Field fr.lifl.adaj.PaquetVars vars>
53 dup
54 invokevirtual #213 <Method int intVar()>
57 istore_1
58 dup
59 invokevirtual <Method int intVar()>
62 istore_2
63 pop
64 goto 114
67 iconst_0
68 istore_1
69 goto 201
72 nop
73 aload_0
74 getfield #190 <Field int etat>
77 ifeq 104
80 aload_0
81 iconst_0
82 putfield #193 <Field int no>
85 nop
96 new #195 <Class fr.lifl.adaj.PaquetVars>
89 dup
90 iconst_1
91 invokespecial #196 <Method fr.lifl.adaj.PaquetVars(int)>
94 dup
95 iload_1
96 invokevirtual #201 <Method void addVar(int)>
99 aload_0
100 swap
101 putfield <Field fr.lifl.adaj.PaquetVars vars>
104 aload_0
105 invokevirtual #216 <Method void migrer()>
108 nop
109 iconst_0
110 istore_2
111 goto 192
114 nop
115 aload_0
```

**116 getfield #190 <Field int etat>**
**119 ifeq 155**
**122 aload_0**
**123 iconst_1**
**124 putfield #193 <Field int no>**
**127 nop**
**128 new #195 <Class fr.lifl.adaj.PaquetVars>**
**131 dup**
**132 iconst_2**
**133 invokespecial #196 <Method fr.lifl.adaj.PaquetVars(int)>**
**136 dup**
**137 iload_1**
**138 invokevirtual #201 <Method void addVar(int)>**
**141 dup**
**142 iload_2**
**143 invokevirtual #201 <Method void addVar(int)>**
**146 aload_0**
**147 swap**
**148 putfield #205 <Field fr.lifl.adaj.PaquetVars vars>**
**151 aload_0**
**152 invokevirtual #216 <Method void migrer()>**
**155 nop**
156 getstatic #2 <Field java.io.PrintStream out>
159 new #3 <Class java.lang.StringBuffer>
162 dup
163 invokespecial #4 <Method java.lang.StringBuffer()>
166 iload_1
167 invokevirtual #5 <Method java.lang.StringBuffer append(int)>
170 ldc #6 <String " ">
172 invokevirtual #7 <Method java.lang.StringBuffer append(java.lang.String)>
175 iload_2
176 invokevirtual #5 <Method java.lang.StringBuffer append(int)>
179 ldc #6 <String " ">
181 invokevirtual #7 <Method java.lang.StringBuffer append(java.lang.String)>
184 invokevirtual #8 <Method java.lang.String toString()>
187 invokevirtual #9 <Method void print(java.lang.String)>
190 iinc 2 1
193 iload_2
194 iconst_3
195 if_icmplt 114
198 iinc 1 1
201 iload_1
202 iconst_3
203 if_icmplt 72
206 return

# Annexe F

# Script de lancement des applications ADAJ

Le script suivant concerne le lancement des tests pour une certaine distribution (donnée comme troisième argument), sur un ensemble de machines, chaque machine donnée par une ligne d'un fichier, passé comme deuxième argument, et dont le RuntimeManager tourne sur la machine donnée comme premier argument. Les paramètres de l'application sont la taille d'une sous-population, le nombre d'itérations, le nombre de cycles (les trois paramètres suivants). Les paramètres de la stratégie de distribution :

- la période de lissage et
- la fréquence de la politique d'informations de charge,

constituent les dernières arguments.

### L'exécutable *scriptapp*

```
distr=$3
taille=$4
it=$5
lgcycle=$6
relTime=$7
inspTime=$8

cd ${HOME}/newloadtest/${taille}.${it}.${distr}/a20
${HOME}/newloadtest/script $1 $2 20 0 ${distr} ${taille} ${it} ${lgcycle}
                              ${relTime} ${inspTime}

for algo in 30 40 50
do
  for seuil in 0.088 0.1 0.3 0.5
  do
      cd ${HOME}/newloadtest/${taille}.${it}.${distr}/a${algo}s${seuil}
      ${HOME}/newloadtest/script $1 $2 ${algo} ${seuil} ${distr}
                        ${taille} ${it} ${lgcycle} ${relTime} ${inspTime}
  done
done
```

Le script appelé crée l'environnement distribué par l'ajout successif de machines, et lance ensuite l'application avec les arguments donnés. Les sorties correspondantes à chaque machine sont respectivement redirigées dans des fichiers.

**L'exécutable *script***

```
pwd='pwd'
hostm=$1
filename=$2
algo=$3
seuil=$4
distr=$5
taille=$6
lgcycle=$7
it=$8
relTime=$9
inspTime=$10

rsh -2 -f $hostm " cd $pwd  ; jprm -verbose -host $hostm &"
sleep 20

# lancement des machines
for m in 'cat ${HOME}/newloadtest/${filename}'
do
   rsh -2 -f $m " cd $pwd  ; jpvm -verbose -host $hostm > m$m &"
   sleep 10
done
# lancement de l'application
time jadaj -host $hostm -relation:sleepTime=${relTime}
           -lb:${algo}:${seuil}:inspTime=${inspTime} tspjpcgg.TspDistr 0
           ${HOME}/applic/TSPdata/eil51.tsp ${taille} ${lgcycle}
           ${HOME}/newloadtest/${distr} ${it}

jprk -host $hostm
```

Un exemple de lancement de script, pour des sous-populations de taille 200, un nombre de 35 itérations, un nombre de cycles de 30, dans une distribution donnée par le fichier d2.dat, est l'exemple suivant :

```
cat param.txt | ./scriptapp
```

où `param.txt` est un fichier contenant la ligne

```
if03 filehost d2.dat 200 35 30 5000 8000
```

et

- $if03$ est la machine où est lancé le composant central,
- les machines de calcul sont données par le fichier *filehost*, et
- les paramètres de la stratégie de distribution sont
  - $t$=5000 ms la période de lissage, des informations de relations, et
  - $\Delta t$ = 8000 ms la fréquence de la politique d'information de charge.

# Annexe G

# Résultats des évaluations du mécanisme d'équilibrage ADAJ

| $gain_b$/ noobs+nolb(%) | algo20 | seuil | algo30 | algo40 | algo50 |
|---|---|---|---|---|---|
| d2.dat | 27,03 | s=0,088 | 28,28 | 31,42 | 30,38 |
| | | s=0,1 | 29,53 | 30,91 | 30,97 |
| | | s=0,3 | 29,24 | 30,77 | 31 |
| | | s=0,5 | 28,14 | 27,34 | 29,72 |
| ddes.dat | 14,6 | s=0,088 | 25,83 | 26,57 | 27,22 |
| | | s=0,1 | 25,15 | 26,21 | 26,33 |
| | | s=0,3 | 26,62 | 26,68 | 28,01 |
| | | s=0,5 | 22,49 | 26,15 | 24,38 |
| ddes2.dat | 23,53 | s=0,088 | 28,58 | 31,32 | 31,14 |
| | | s=0,1 | 26,64 | 31,19 | 29,51 |
| | | s=0,3 | 23,25 | 28,68 | 29,77 |
| | | s=0,5 | 27 | 26,92 | 28,45 |
| dseul.dat | 52,26 | s=0,088 | 54,11 | 53,58 | 53,32 |
| | | s=0,1 | 56,77 | 57,71 | 57,74 |
| | | s=0,3 | 49,64 | 51,71 | 56,73 |
| | | s=0,5 | 48,35 | 48,23 | 54,71 |

Tableau G.1 – Gains bruts (en %) obtenus par les distributions de charge (taille 200, itérations 35)

# Bibliographie


[ABL+95] Arabe (J.), Beguelin (A.), Lowekamp (B), Seligman (E.), Starkey (M.) et Stephan (P.). – *Dome : Parallel programming in a heterogenous multi-user environment.* – Rapport technique, Carnegie Mellon University, Avril 1995.

[AF89] Artsy (Yeshayahu) et Finkel (Raphael A.). – Designing a Process Migration Facility : The Charlotte Experience. *IEEE Computer*, vol. 22, n° 9, 1989, pp. 47–56.

[AK96] Andre (D.) et Koza (J.R.). – *Parallel Genetic Programming : A Scalable Implementation using Transputer Network Architecture - Advances in Genetic Programming*, chap. 16, pp. 317–337. – Angeline, P. and Kinnear Jr, K.E., 1996.

[APGG00] Amiri (Khalil), Petrou (David), Ganger (Gregory R.) et Gibson (Garth A.). – Dynamic Function Placement for Data-intensive Cluster Computing. *In : USENIX Annual Technical Conference* . – San Diego, USA, CA, 2000.

[BAAD91] Babaoglu (O.), Alvisi (L.), Amoroso (A.) et Davoli (R.). – Mapping Parallel Computations onto Distributed Systems in Paralex. *In : Proceedings IEE CompEuro'91 Conference*, pp. 123–130. – Bologna, Italy, 1991.

[Bad00] Badidi (Elarbi). – *Architecture et services pour la distribution de charge dans les systèmes distribués objet.* – Thèse de PhD, Université de Montréal, faculté des Arts et des Sciences - Départment d'Informatique et de Recherche Opérationnelle, 2000.

[BB97] Brase (C.H.) et Brase (C.P.). – *Understanding Basic Statistics.* – Houghton Mifflin Company, 1997.

[BB99] Baker (Mark) et Buyya (Rajkumar). – *High Performance Cluster Computing : Architectures and Systems*, chap. Cluster Computing at a Glance, pp. 1–47. – Prentice Hall, 1999.

[BBI+94] Banâtre (Michel), Belhamissi (Yasmina), Issarny (Valérie), Puaut (Isabelle) et Routeau (Jean-Paul). – *Adaptive Placement of Method Executions within a Customizable Distributed Object-Based Runtime System - Design, Implementation, and Performance.* – ISSN 1350-2042 n° TR 64, IRISA and CRIN-Nancy, 1994.

[BcBC02] Baduel (Laurent), coise Baude (Fran) et Caromel (Denis). – Efficient, flexible, and typed group communications in Java. *In : Joint ACM-ISCOPE conference on Java Grande - ISCOPE.* – Seattle, Washington, November 2002.

[BCHV00] Baude (F.), Caromel (D.), Huet (F.) et Vayssière (J.). – Communicating Mobile Objects in Java. *HPCN 2000*, vol. LNCS 1823, 2000, pp. 633–643.

[BCKP95] Bagrodia (R.), Chu (W.), Kleinrock (L.) et Popek (G.). – Vision, Issues, and Architecture for Nomadic Computing. *IEEE Personal Communications*, December 1995, pp. 14–27.

[BF81] Bryant (R.) et Finkel (R.). – A Stable Distributed Scheduling Algorithm. *In : 2nd International Conference on Distributed Computing Systems*, pp. 314–323.

[BH00] Bouchenak (Sarah) et Hagimont (D.). – Approaches to Capturing Java Threads State. *In : IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000).* – New York - USA, 2000.

[BH02] Bouchenak (S.) et Hagimont (D.). – *Zero Overhead Java Thread Migration.* – Rapport technique, Institut National de Recherche en Informatique et en Automatique, 2002.

[BK99] Brunner (Robert K.) et Kalé (Laxmikant V.). – Adapting to Load on Workstation Clusters. *In : The Seventh Symposium on the Frontiers of Massively Parallel Computation (Frontiers'99).* pp. 106–112. – Annapolis, USA, Février 1999.

[BLL00] Bouchi (A.), Leprêtre (E.) et Lecouffe (P.). – Un mécanisme d'observation des objets distribués en Java. *In : Rencontres Francophones du Parallélisme des Architectures et des Systèmes (RenPar'12)*, pp. 171–176. – Besançon, France, Avril 2000.

[BLS99] Barak (Amnon), La'adan (Oren) et Shiloh (Amnon). – Scalable Cluster Computing with MOSIX for LINUX. *In : Proc. 5-th Annual Linux Expo*, pp. 95–100. – Raleigh, N.C., 1999.

[BOT01] Bouchi (A.), Olejnik (R.) et Toursel (B.). – Java Tools for Measurements of the Machines Loads. *In : "NATO Advanced Research Workshop Romania - Advanced Environments, Tools and Applications for Cluster Computing".* – Mangalia, Roumanie, Septembre 2001.

[BOT02] Bouchi (A.), Olejnik (R.) et Toursel (B.). – A new estimation for distributed Java object activity. *In : Proceedings of International Parallel and Distributed Processing Symposium (IPDPS).* – Fort Lauderdale, Florida, USA, Avril 2002.

[Bou98] Bouchenak (S.). – *Mécanismes pour la migration de processus. Extension de la machine virtuelle Java.* – Mémoire de DEA, Institut National Polytechnique de Grenoble - ENSIMAG, 1998.

[Bou99] Bouchenak (Sarah). – Pickling threads state in the Java system. *In : Third European Research Seminar on Advances in Distributed Systems (ERSADS'99).* – Madeira Island, Portugal, 1999.

[Bou01] Bouchenak (S.). – *Mobilité et Persistenance des Applications dans l'Environnement Java.* – Thèse de PhD, L'Institut National Polytechnique de Grenoble, 2001.

[Bou03] Bouchi (A.). – *Proposition d'un mécanisme d'observation dynamique de l'exécution d'applications Java distribuées.* – Thèse de PhD, Université des Sciences et Technologies de Lille, 2003.

[Bub96] Bubendorfer (Kristian Paul). – *Resource Based Policies for Load Distribution.* – Thèse de PhD, Victoria University of Wellington, 1996.

[Cah00] Cahon (S.). – *Une métaheuristique hybride et distribuée avec Java. Application au problème de voyageur de commerce.* – Rapport de stage, LIFL, 2000.

[Cav99] Cavalheiro (Gerson G.H.). – *ATHAPASCAN-1 : Interface générique pour l'ordonnancement dans un environnement d'exécution parallèle.* – Thèse de PhD, Institut National Polytechnique de Grenoble, 1999.

[cBCH+02] coise Baude (Fran}, Caromel (Denis), Huet (Fabrice), Mestre (Lionel) et Vayssière (Julien). – Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. *In : HPDC-11*, pp. 93–102. – Edinburgh, Ecosse, 2002.

[CHPB96] Chatonnay (Pascal), Herrmann (Bénédicte), Philippe (Laurent) et Bourdon (François). – Placement dynamique dans les systèmes répartis à objets. *Calculateurs parallèles*, vol. 8, n° 1, 1996, pp. 11–30.

[CK88] Casavant (Thomas L.) et Kuhl (Jpn K.). – A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, vol. 14, n° 2, 1988.

[CKV98] Caromel (D.), Klauser (W.) et Vayssière (J.). – Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, vol. 10, 1998.

[CLL85] Carey (M.), Livny (M.) et Ly (H.). – Dynamic Task Allocation in a Distributed Database System. *In : 5th International Conference on Distributed Computing Systems.* – Denver, 1985.

[CLZ98] Corradi (Antonio), Leonardi (Letizia) et Zambonelli (Franco). – On the Effectiveness of Different Diffusive Load Balancing Policies in Dynamic Applications. *In : Proceedings of the Conference on High Performance Computing and Networking Europe '98 (HPCN '98)*, éd. par Lecture Notes in Computer Science. pp. 274–283. – Amsterdam, Pays Bas, 1998.

[Col99] Coley (David A.). – *An Introduction to Genetic Algorithms for Scientists and Engineers.* – World Scientific, 1999.

[CWI] CWI - The National Research Institute for Mathematics and Computer Science in the Netherlands. – *Test Set for IVP Solvers.* – http ://www.cwi.nl/ftp/IVPtestset/descrip.htm.

[Dav91] Davis (Lawrence). – *Handbook of genetic algorithms.* – VNR Computer Library, 1991.

[DS ] DS & 0 Research Team. – *Projet Jacob - Active Container of Object for Java.* – http ://jccf.labri.u-bordeaux.fr/jodo/.

[ELZ85] Eagar (D.L.), Lazowska (E.D.) et Zahorjan (J.). – A Comparaison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *ACM SIGMETRICS*, 1985, pp. 53–68.

[ELZ86] Eagar (D.L.), Lazowska (E.D.) et Zahorjan (J.). – Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, vol. SE-12, n° 5, 1986, pp. 662–675.

[FBCL91] Feeley (M.J.), Bershad (B.N.), Chase (J.S) et Levy (H.M.). – Dynamic node reconfiguration in a parallel distributed environment. *In : Proceedings of the third ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 114–121. – Williamsburg, Virginia, United States, 1991.

[FDLT01] Felea (V.), Devesa (N.), Lecouffe (P.) et Toursel (B.). – Expressing Parallelism in Java Applications Distributed on Clusters. *In : Advanced Research Workshop Romania 2001 - Advanced Environments, Tools and Applications for Cluster Computing.*

[Fel02] Felea (V.). – Exploiting Runtime Information in Load Balancing Strategies. *In : Distributed and Parallel Systems - Cluster and Grid Computing*, éd. par P. Kacsuk and D. Kranzlmüller and Z. Németh and J. Volkert. pp. 21–29. – Linz, Austria, 2002.

[Fio01] Fiolet (Valerie). – *Data Mining Distribué.* – Rapport de stage, LIFL, 2001.

[Fis78] Fishburn (P.C). – A survey of multiattribute/multicriteria evaluation theories. *In : Multicriteria problem solving*, éd. par Zionts (S.), pp. 181–224. – Springer Verlag, Berlin, 1978.

[FK00] Foster (Ian) et Kesselman (Carl). – Computational Grids. *In : VECPAR'2000 - 4th International Meeting on Vector and Parallel Processing.* – Porto, Portugal, 2000.

[Fol92] Folliot (Berthil). – *Méthodes et outils de partage de charge pour la conception et la mise-en-œuvre d'applications dans les systèmes répartis.* – Thèse de PhD, Institut Blaise Pascal, 1992.

[For03] Forum (Message Passing Interface). – *The Message Passing Interface (MPI) standard.* – http ://www-unix.mcs.anl.gov/mpi/, 1995-2003.

[FR97] Frank (E.) et Redmond (I.). – *DCOM : Microsoft Distributed Component Object Model.* – IDG Books worldwides, 1997.

[FT02] Felea (V.) et Toursel (B.). – Methodology for Java Distributed and Parallel Programming Using Distributed Collections. *In : Proceedings of International Parallel and Distributed Processing Symposium (IPDPS).* – Fort Lauderdale, Florida, USA, Avril 2002.

[FT03] Felea (Violeta) et Toursel (Bernard). – Middleware-based Load Balancing for Communicating Java Objects. *In : Proceedings Concurrent Information Processing and Computing (CIPC) NATO ARW.* – Sinaia, Romania, 2003.

[FTD01] Felea (V.), Toursel (B.) et Devesa (N.). – Les collections distribuées : un outil pour la conception d'applications Java parallèles. *In : Rencontres Francophones du Parallélisme des Architectures et des Systèmes (RenPar'13)*, pp. 97–102. – Paris, France, Avril 2001.

[FTD03] Felea (V.), Toursel (B.) et Devesa (N.). – Les collections distribuées : un outil pour la conception d'applications Java parallèles. *Technique et science informatiques*, vol. 22, n° 3, 2003, pp. 289–314.

[Fün98] Fünfrocken (Stefan). – Transparent Migration of Java-based Mobile Agents. *In : Proceedings of 2nd International Workshop on Mobile Agents, MA1998*, pp. 26–37. – Stuttgart, Allemagne, 1998.

[FZ87] Ferrari (Domenico) et Zhou (Songnian). – *An Empirical Inverstigation of Load Indices for Load Balancing Applications.* – Rapport technique n° CSD-87-353, University of California at Berkeley, Janvier 1987.

[Gol94] Goldberg (David E.). – *Algorithmes génétiques : exploration, optimisation et apprentissage automatique.* – Addison-Wesley France, 1994.

[Gru69] Grubbs (F.E.). – Procedures for detecting outlying observations in samples. *Technometrics*, vol. 11, n° 1, 1969, pp. 1–21.

[GT98] Gaber (J.) et Toursel (B.). – Dynamic and Randomized Load Distribution in Arbitrary Networks. *In : EuroPar.* – Soupthampton, Anlgeterre, 1998.

[Har86] Harbus (R.S.). – *Dynamic process migration : to migrate or not to migrate.* – Rapport technique n° CSRI-42, University of Toronto, Canada, 1986.

[Hue02] Huet (F.). – *Objets mobiles : conception d'un middleware et évaluation de la communication.* – Thèse de PhD, Université de Nice - Sophia Antipolis, 2002.

[HW79] Hartigan (J. A.) et Wong (M. A.). – A K-Means Clustering Algorithm. *Applied Statistics*, vol. 28, 1979, pp. 100–108.

[IBM] IBM Tokyo Research Labs. – *Aglets Workbench : Programming Mobile Agents in Java 1996.* – http ://www.trl.ibm.co.jp/aglets.

[ICB99] Izatt (Matthew), Chan (Patrick) et Brecht (Tim). – Ajents : Towards an Environment for Parallel, Distributed and Mobile Java Applications. *In : ACM 1999 Java Grande Conference.* – San Francisco, California, Juin 1999.

[IGD+97] Ivannikov (V.), Gaissaryan (S.), Domrachev (M.), Etch (V.) et Shtaltovnaya (N.). – *DPJ : Java class library for development of data-parallel programs.* – Institute for System Programming, Russian Academy of Sciences , 1997.

[Jen96] Jensen (Christian Damsgaard). – Fine-Grained Load Distribution in Object Based Systems An Experiment with Grain Sizes in Guide-2. *Calculateurs parallèles*, vol. 8, n° 1, 1996, pp. 31–48.

[KG94] Kumar (V. P.) et Gupta (A.). – Analyzing Scalability of Parallel Algorithms and Architectures. *Journal of Parallel and Distributed Computing*, vol. 22, n° 3, 1994, pp. 379–391.

[KGA+00] Kon (Fabio), Gill (Binny), Anand (Manish), Campbell (Roy H.) et Mickunas (M. Dennis). – Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents. *In : ASA/MA*, pp. 86–98. – Zurich, Suisse, 2000.

[Kun91] Kunz (Thomas). – *The Influence of Different Load Descriptions on a Heuristic Load Balancing Scheme.* – Rapport technique n° TI-6/91, Institut für Theoretische Informatik, Technische Hochschule Darmstadt, Decembre 1991.

[LK87] Lin (F.) et Keller (R.). – The Gradiant Model Load Balancing Method. *IEEE Transactions on Software Engineering*, vol. 13, n° 1, 1987.

[LLM88] Liztkow (Michael J.), Livny (Miron) et Mutka (Matt W.). – Condor - A Hunter of Idle Workstations. *In : Proceedings 8th International Conference on Distributed Computing Systems*, pp. 104–111. – San Jose, 1988.

[LP00] Launay (Pascale) et Pazat (Jean-Louis). – Écrire parallèle, exécuter distribué. *Technique et science informatique*, vol. 19, n° 9, 2000, pp. 1193 – 1221.

[MDLT96] Melab (N.), Devesa (N.), Lecouffe (M.P.) et Toursel (B.). – Adaptive load balancing of irregular applications. A case study : IDA* applied to the 15-puzzle problem. *In : Proc. of the Third Intl. Workshop, IRREGULAR'96.* pp. 327–338. – Santa Barbara, California, USA, 1996.

[Mel97] Melab (N.). – *Gestion de la granularité et régulation de charge dans le modèle P3 d'évaluation parallèle des langages fonctionnels.* – Thèse de PhD, LIFL, Université de LilleI, 1997.

[MLC98] Milojicic (Dejan S.), LaForge (William) et Chauhan (Deepika). – Mobile Objects and Agents. *Distributed System Engineering*, vol. 5, 1998, pp. 214–227.

[MNV+99] Maassen (J.), Nieuwpoort (R.), Veldema (R.), Bal (H.) et Plaat (A.). – An Efficient Implementation of Java's Remote Method Invocation. *In : ACM Symposium on Principle and Practice of Parallel Programming (PPOPP)*, pp. 173–182. – Atlanta, Georgia, USA, Juillet 1999.

[MvRT+90] Mullender (Sape J.), van Rossum (Guido), Tanenbaum (Andrew S.), van Renesse (Robbert) et van Staveren (Hans). – Amoeba : A Distributed Operating System for the 1990s. *IEEE Computer - Special Issue : Recent Developments in Operating Systems*, vol. 23, n° 5, 1990, pp. 44–53.

[MWL99] Ma (Matchy J.M.), Wang (Cho-Li) et Lau (Francis C.M.). – Delta Execution : A Preemptive Java Thread Migration. *In : International Conference on Parallel and*

*Distributing Processing Techniques and Applications (PDPTA99)*, pp. 518–524. – Las Vegas, USA, 1999.

[MZD92] Milojicic (W.), Zint (W.) et Dangel (A.). – *Task Migration on Top of the Mach Microkernel - Design and Implementation -*. – Rapport technique, Kaiserslautern, 1992.

[NM95] Namyst (R.) et Méhaut (J.F.). – $Pm^2$ : Parallel multithreaded machine. a computing environment for distributed architectures. *In : Parco'95*, pp. 279–285. – Gent, Belgique, 1995.

[NPH99] Nester (C.), Philippsen (M.) et Haumacher (B.). – A More Efficient RMI for Java. *In : ACM 1999 Java Grande Conference*, pp. 153–159. – San Francisco, CA, Juin 1999.

[Obj96] ObjectSpace. – *Voyager - JGL libraries.* – http ://www.objectspace.com/, release version 3.1 - 1996.

[OMG] *Object Management Group* . – http ://www.corba.org.

[OTCH94] O'Connor (Martin), Tangney (Brendan), Cahill (Vinny) et Harris (Neville). – Microkernel Support for Migration. *Distributed Systems Engineering Journal*, no4, 1994, pp. 212–223.

[OW00] Oaks (Scott) et Wong (Henry). – *Java Threads.* – O'Reilly, 2000.

[PBR99] Palma (N. De), Bellissard (L.) et Riveill (M.). – Dynamic Reconfiguration of Agent-Based Applications. *In : Third European Research Seminar on Advances in Distributed Systems (ERSADS'99).* – Madeira Island - Portugal, Avril 1999.

[Per97] Perret (S.). – *Agents mobiles pour l'accès nomade à l'information répartie dans les réseaux de grande envergure.* – Thèse de PhD, Université Joseph Fourrier, 1997.

[Phi00] Philippe (Laurent). – Administration de l'exécution de composants logiciels sur les plate-formes à objets répartis, 2000. Thèse pour obtenir le diplôme d'habilitation à diriger les recherches de l'Université de Franche-Comté.

[PM83] Powell (Michael L.) et Miller (Barton P.). – Process Migration in DEMOS/MP. *ACM Operating Systems Review*, vol. 17, n° 5, 1983, pp. 110–119.

[PZ97] Phillippsen (M.) et Zenger (M.). – JavaParty - Transparent Remote Objects in Java. *In : ACM 1997 Workshop on Java for Science and Engineering Computation.* – Las Vegas, USA, Juin 1997.

[Rac97] Rackl (Günter). – *Load Distribution for CORBA Environments.* – Thèse, Technische Universität München - Institut für Informatik, 1997. Master Thesis.

[RADS97] Ranganathan (M.), Acharya (A.), Dharma (S.) et Saltz (J.). – Network-aware Mobile Programs. *In : Proceedings of the USENIX 1997 Annual Technical Conference*, pp. 1–13. – Anaheim, California, USA, Janvier 1997.

[RD00] Roussel (Gilles) et Duris (Etienne). – *Java et internet : concepts et programmation.* – Paris : Vuibert Informatique, 2000.

[RM90] Ross (A.) et McMillin (B.). – Experimental comparaison of bidding and drafting load sharing protocols. *In : Proceedings of the 6th Distributed Memory Computing Conference*, pp. 968–974. – Portland, USA, 1990.

[SGG01] Silberschatz (A.), Galvin (P.) et Gagne (G.). – *Principes appliqués des systèmes d'exploitation : avec Java.* – Vuibert Informatique, Paris, 2001.

[SH98] Smith (Peter) et Hutchinson (Norman C.). – Heterogeneous Process Migration : The Tui System. *Software - Practice & Experience*, vol. 28, n° 6, 1998, pp. 611–639.

[Shu90] Shub (C.S). – Native Code Process-Originated Migration in a Heterogeneous Environment. *In : Proceedings of the 1990 ACM Annual Computer Science Conference*, pp. 266–270. – Washington, février 1990.

[SKK99a] Schloegel (Kirk), Karypis (George) et Kumar (Vipin). – A New Algorithm for Multi-objective Graph Partitioning. *In : Proceedings of European Conference on Parallel Processing, Europar'99.* pp. 322–331. – Toulouse, France, 1999.

[SKK99b] Schloegel (Kirk), Karypis (George) et Kumar (Vipin). – *Graph Partitioning for High Performance Scientific Simulations.* – Rapport technique n° TR 00-018, Department of Computer Science and Engineering, University of Minnesota, Mineapolis, 1999.

[SMY99] Sekiguchi (Tatsurou), Masuhara (Hidehiko) et Yonezawa (Akinori). – A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. *In : Proceedings of Third International Conference on Coordination Models and Languages - Coordination Languages and Models 1999*, pp. 211–226. – Amsterdam, Pays Bas, Avril 1999.

[SSY00] Sakamoto (Takahiro), Sekiguchi (Tatsurou) et Yonezawa (Akinori). – Bytecode Transformation for Portable Thread Migration in Java. *In : Proceegings of Second International Workshop Mobile Agents (MA2000).* – Zurich, Suisse, 2000.

[Sue00] Suezawa (T.). – Persistent Execution State of a Java Virtual Machine. *In : ACM 2000 Java Grande Conference.* – San Francisco, USA, 2000.

[SUNa] *Java 2 Platform - Std. Ed. v1.3.1.* – http ://java.sun.com/j2se/1.3/docs/api/java/rmi/server/RMIClassLoader.html.

[SUNb] *Sun Products - Remote Method Invocation JDK1.2.* – http ://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html.

[SUNc] *Sun Products - Collections JDK1.2.* – http ://java.sun.com/products/jdk/1.2/docs/guide/collections/index.html.

[Sund] Sun Products - Java Language Specification. – *Method Invocation Expressions.* – http ://java.sun.com/docs/books/jls/second_edition/html/expressions.doc.html#20448.

[Sune] Sun Products - JDK1.2. – *Java Core Reflection.* – http ://java.sun.com/products/jdk/1.2/docs/guide/reflection/spec/java-reflectionTOC.doc.html.

[Sunf] Sun Products - JDK1.2. – *Java Tutorial - Synchronizing Threads.* – http ://java.sun.com/docs/books/tutorial/essential/threads/multithreaded.html.

[Sung] Sun Products - JDK1.2. – *Object Serialization.* – http ://java.sun.com/j2se/1.3/docs/guide/serialization/index.html.

[Sunh] Sun Products - JDK1.2. – *The Java Language Specification.* – http ://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html.

[Suni] Sun Products - JDK1.2. – *Understanding Thread Priority.* – http ://java.sun.com/docs/books/tutorial/essential/threads/priority.html.

[Sunj] Sun Products - JDK1.4. – *Java Regular Expressions.* – http ://java.sun.com/j2se/1.4/docs/api/java/util/regex/package-summary.html.

[Sve90] Svensson (Anders). – History, an Intelligent Load Sharing Filter. *In : Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, pp. 546–553. – Washington DC, USA, 1990.

[SW65] Shapiro (S.S.) et Wilk (M.B.). – An analysis of variance test for normality (complete samples). *Biometrika*, vol. 52, 1965, pp. 591–611.

[Tal97] Talbi (E.G.). – Une taxonomie des algorithmes d'allocation dynamique de processus dans les systèmes parallèles et distribués. *In : Proceedings ISYPAR'97 - 2ème Ecole d'Informatique des Systèmes Parallèles et Répartis*, pp. 136–164. – Toulouse, France, 1997.

[TC92] Tangney (Brendan) et Condon (Andrew). – Some Issues in Load Balancing in Amadeus. *In : Proceedings ECOOP'92 Workshop on Load Balancing In Object Oriented Systems.* – Utrecht, Pays Bas, 1992.

[Tea02] Team (ProActive). – *ProActive - The Java library for Parallel, Distributed, Concurrent computing with Security & Mobility.* – http ://www-sop.inria.fr/sloop/javall/index.html, version 0.9.3 - 2002.

[TLC85] Theimer (Marvin M.), Lantz (Keith A.) et Cheriton (David R.). – Preemptable remote execution facilities for the V-system. *ACM SIGOPS Operating Systems Review*, vol. 19, n° 5, 1985, pp. 2–12.

[TRC+00] Truyen (Eddy), Robben (Bert), Coninx (Tim), Joosen (Wouter) et Verbataen (Pierre). – Portable Support for Transparent Thread Migration in Java. *In : Proceedings of Mobile Agents (MA2000).* – Zürich, Suisse, 2000.

[VDL98] Verbièse (L.), Devesa (N.) et Lecouffe (P.). – ACADA : Aide à la Conception et à l'Exécution d'Applications Distribuées Adaptatives. *In : Rencontres Francophones du Parallélisme des Architectures et des Systèmes (RenPar'10).* – Strasbourg, France, 1998.

[Vig99] Vignéras (Pierre). – Jacob : un support Java pour la distribution et le calcul. *Réseaux et systèmes répartis*, vol. 12, 1999.

[vLSM00] von Laszewski (Gregor), Shudo (Kazuyuki) et Muraoka (Yoichi). – Grid-based Asynchronous Migration of Execution Context in Java Virtual Machines. *In : European Conference on Parallel Computing (EuroPar2000)* . – Munich, Allemagne, 2000.

[WA99] Wilkinson (Barry) et Allen (Michael). – *Parallel Programming. Techniques and Applications using Networked Workstations and Parallel Computers.* – Upper Saddle River, New Jersey 07458, Prentice Hall, 1999.

[WTV02] Weyns (Danny), Truyen (Eddy) et Verbaeten (Pierre). – Distributed Threads in Java. *In : Proceedings of International Symposium on Distributed and Parallel Computing (ISDPC'02).* – Iasi, Roumanie, 2002.

[Zak99] Zaki (Mohamed J.). – Parallel and Distributed Associated Mining : A Survey. *IEEE Concurrency*, 1999, pp. 14–25.

[Zay87] Zayas (Edward R.). – Attacking the Process Migration Bottleneck. *In : 11th Symposium on Operating Systems Principles.* pp. 13–22. – Austin, USA, 1987.

[ZG99] Zalzala (A.M.S.) et Green (D.). – A Multithreaded Java Tool for Genetic Programming Applications. *In : Proceedings of Congress on Evolutionary Computing.* – Washington DC, USA, 1999.

[Zho88] Zhou (Songnian). – A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, no14, 1988, pp. 1327–1341.