*N° d'ordre : 368*
*N° attribué par la bibliothèque : 06ENSL0 368*

**ÉCOLE NORMALE SUPÉRIEURE DE LYON**
Laboratoire de l'Informatique du Parallélisme

## THÈSE

*pour obtenir le grade de*
**Docteur de l'École Normale Supérieure de Lyon**
**spécialité : Informatique**

*au titre de l'école doctorale de MathIF*

*présentée et soutenue publiquement le 11 octobre 2006*

par Monsieur Antoine VERNOIS

# Ordonnancement et réplication de données bioinformatiques dans un contexte de grille de calcul

| | | | |
|---|---|---|---|
| Directeurs de thèse : | Monsieur | Frédéric | DESPREZ |
| | Monsieur | Christophe | BLANCHET |
| Après avis de : | Monsieur | Vincent | BRETON, Rapporteur |
| | Monsieur | Michel | DAYDÉ, Rapporteur |

Devant la commission d'examen formée de :

| | | |
|---|---|---|
| Monsieur | Christophe | BLANCHET, Membre |
| Monsieur | Vincent | BRETON, Membre/Rapporteur |
| Monsieur | Michel | DAYDÉ, Membre/Rapporteur |
| Monsieur | Frédéric | DESPREZ, Membre |
| Monsieur | Dominique | LAVENIER, Membre |
| Madame | Pascale | VICAT-BLANC PRIMET, Membre |

# Merci

Dans une thèse, en plus du travail, il y a deux choses réellement importantes, le pot qui suit la soutenance et la section des remerciement par laquelle débute le manuscrit. La soutenance est passée, le pot aussi. Il était vraiment réussi, mais je n'ai pas de mérite, je l'avais honteusement délégué à mes parents qui ont fait quelque chose de vraiment bien, comme toujours. Les remerciements, par contre, c'est moi qui m'en charge. Pas question que je les confie à quelqu'un d'autre. C'est la synthèse, du point de vue humain, de l'aventure que représente une thèse.

Comment pourrais-je commencer par quelqu'un d'autre que Frédéric Desprez, Fred, mon directeur de thèse. Initialement, il ne devait avoir qu'un rôle minimal dans cette thèse, mais je ne pourrais jamais le remercier assez d'avoir repris les choses en main lorsque je pataugeais, d'y avoir cru beaucoup plus que moi, d'avoir continué à me faire confiance quand ma productivité frôlait le zéro. Sans oublier bien sûr, les scéances de natation, et les mythiques concerts de SOAD et Korn !

Je ne peux pas non plus oublier Christophe Blanchet qui est à l'origine du sujet de cette thèse et Pascale Primet qui m'a convaincu que les bioinformaticiens avaient des problématiques qui méritaient qu'on y consacre quelques années.

Je tiens aussi à remercier Vincent Breton et Michel Daydé pour leur travail de rapporteur ainsi que Dominique Lavenier d'avoir accepté de présider mon jury.

Je pense à tous ceux que j'ai croisé au fil de ces années au LIP et durant mon bref passage à l'IBCP, et qui m'ont permis de travailler dans de bonnes et agréables conditions. Anne-Pascale, Corine, Isabelle, Sylvie et Christine, les secrétaires du LIP et de l'IBCP, aussi gentilles qu'efficaces alors même que je venais faire mes ordres de mission à la dernière minute. Merci à la bioinfo-team de l'IBCP qui m'a fait une formation expresse en biologie et bioinformatique et a supporté mon caractère d'informaticien. J'ai aussi une pensée particulière pour un membre à part entière de ce labo, qu'on oublie souvent : la machine à café.

Et puis, il y a tous mes co-bureaux et voisins de bureau, tous les membres de l'équipe GRAAL au cours du temps. Le clan des chefs, Fred bien sûr, mais aussi Yves, Eddy, Frédo, Jean-Yves, toujours à l'écoute, rarement les derniers pour lancer des vannes et appuyer là où ça fait mal mais aussi, et surtout, pour soutenir et encourager. Les anciens, ceux qui ont fini avant nous, nous laissant assumer la lourde tâche de reprendre le statut «d'aînés» auprès des nouveaux : Martin, Arnaud, Abdou. Merci à ceux qui m'ont supporté le plus dans leur bureau, Loris d'abord, pour les discussions autour de la machine à café, les noms d'oiseaux dans les parties de frozen-bubble. Un énorme merci à Hélène. Je crois que si je devais faire la liste de ce que je te dois, j'y serais encore... Alors merci, merci pour tout. Tu resteras à jamais ma *ptite co-bureau préférée* même si l'on ne se retrouvera probalement plus jamais dans le même bureau ;-) Merci aussi à Raphaël, pour les branlés que je lui ai mises à frozen-bubble et sa façon systématique de se vexer, mais d'en redemander le lendemain :-) Et puis tous les ptis jeunes qui sont arrivés après, Cedric, Emmanuel, Jean-François, Jean-Sébastien, Vero-

nika, et qui portent maintenant le flambeau. J'allais presque oublier Gil Utard avec qui j'ai fait mon stage de DEA au LIP avant de repartir sous son (absence de) soleil amiénois et qui m'a appris toutes les bases de la recherche scientifique. J'espère que dans un avenir proche j'aurais la chance de recroiser la route de tous ces gens et que l'aventure scientifique pourra continuer.

Il y a aussi ceux qui m'ont supporté en dehors du labo, dans la vraie vie. Mes parents, tout d'abord, à qui je dois beaucoup, pour m'avoir encouragé aveuglément, sans trop savoir ce que je faisais au juste, qui ont accepté mes sautes d'humeur et mes périodes prolongées sans nouvelles sans jamais rien dire. Merci à Anabelle pour nos longues conversations par mail et son soutien de l'autre côté de la Manche. Je suis pas toujours très agréable, j'en ai conscience, et pour continuer à être là sans (trop) poser de questions, merci du fond du cœur. Un grand merci à toute l'équipe d'Impérium Ludi, Benjamin, Marie-Hélène, Rico et les autres, qui m'ont offert des moments de totale évasion quand j'en avais le plus besoin.

# Table des matières

# Table des figures

# Liste des tableaux

# Chapitre 1

# Introduction

## 1.1 La bioinformatique

### 1.1.1 Qu'est-ce-que la bioinformatique ?

La bioinformatique est le domaine des sciences où la biologie, l'informatique, les mathématiques et les technologies de l'information se rejoignent pour former une seule et unique discipline [29]. En cela, elle constitue une nouvelle branche de la biologie en offrant une approche « in silico » qui vient compléter les approches « in situ » (en milieu nature), « in vivo » (dans l'organisme vivant) et « in vitro » (en éprouvette) de la biologie traditionnelle. Même si le terme n'apparaît dans la littérature que dans les années 1990, la bioinformatique fait son apparition dans les années 1980 avec la création des premières banques de données de biomolécules. À ses débuts, la bioinformatique s'intéressait principalement à la création et la maintenance de banques de données permettant le stockage d'information biologique. Le développement de ces banques n'a pas uniquement soulevé des problèmes sur leur structuration mais aussi les moyens offerts à la communauté biologique pour accéder à ses informations, en rajouter de nouvelles ou modifier des entrées existantes. Ces objectifs ont évolué et maintenant la bioinformatique cherche essentiellement à proposer des méthodes et des logiciels qui permettront la gestion, l'analyse, la comparaison ou encore l'exploration d'informations génétiques ou génomiques contenues dans des banques de données. Le but de ces analyses guidées par des outils informatiques est alors la production, et la prédiction, de connaissances nouvelles ainsi qu'élaborer de nouveaux concepts [21]. Pour cela, la bioinformatique a principalement recours à trois types de méthodes élémentaires :

- méthodes comparatives : exploration des données identifiées et annotées pour établir des rapprochements avec des séquences ou structures inconnues.
- méthodes statistiques : analyses statistiques des données pour dégager des règles ou des contraintes.

– approches par modélisation : études des objets visant à construire des modèles tentant d'en extraire les propriétés communes.

Ainsi, la bioinformatique se sert des recherches et des expérimentations déjà menées afin de proposer des voies et des axes pour les prochaines recherches et épargner le coût d'expérimentation « en aveugle ». La bioinformatique, au delà de son caractère initial de gestion de données, est devenue un carrefour pluridisciplinaire capable d'enrichir le domaine fondamental de connaissances nouvelles et d'être à l'origine de concepts biologiques originaux.

### 1.1.2 Les banques de données

La bioinformatique consiste donc essentiellement en un traitement automatique de l'information biologique. En ce sens, les banques de données sont la matière première qui permettra la création de nouvelles connaissances. Les banques de données biologiques sont de larges ensembles de données organisées. Une grande majorité des banques de données sont constituées d'un fichier contenant un grand nombre d'entrées. Chacune de ces entrées est constituée du même ensemble d'information contenant, par exemple, une séquence, sa description, le nom de l'organisme scientifique qui l'a isolée ainsi que des références d'articles associés à cette séquence. La première compilation de séquences de protéines apparaît dès 1965 sous forme d'un atlas imprimé qui contient alors 50 entrées. Il paraîtra uniquement sous forme papier jusqu'en 1978 avant d'être disponible de façon électronique. Avec la découverte des techniques de séquençage, les premières grandes banques généralistes verront le jour au début des années 1980.

Devant la croissance exponentielle et l'hétérogénéité des séquences des banques généralistes, des banques spécialisées se sont constituées autour de thématiques biologiques particulières comme, par exemple, les banques de motifs ou en vue de réunir les séquences d'une même espèce et d'en enrichir les annotations pour diminuer ou lever les ambiguïtés laissées par les grandes banques publiques.

## 1.2 Les grilles de calcul

### 1.2.1 Qu'est ce qu'une grille de calcul ?

Le terme anglophone de « Grid », grille en français, est apparu à la fin des années 1990 [18] et provient de l'analogie avec le fonctionnement des réseaux d'alimentation en électricité, « Electrical Power Grid » en anglais. Les réseaux électriques produisent sans cesse de l'énergie électrique qui est fournie, à la demande, à l'usager qui en a besoin. Ce système est composé de deux grandes parties : d'un coté se trouvent les unités de fabrication d'énergie électrique et de l'autre, il y a les consommateurs. Ces

deux ensembles sont reliés entre eux par un réseau de transport assurant l'acheminement et la régulation de l'énergie. Ainsi, lorsque l'on branche un appareil électrique dans une prise de courant, on a accès à la ressource électricité, sans avoir besoin de savoir ni où ni comment elle a été produite, ni même quel chemin elle a parcouru pour arriver jusqu'à nous. On branche l'appareil dans la prise qui est au mur, et l'appareil électrique peut se mettre en marche.

L'idée des grilles de calcul est semblable à cela à la différence que les ressources fournies aux consommateurs ne sont plus de l'électricité mais de la puissance de calcul, de l'espace de stockage informatique, différents instruments et capteurs ou encore l'accès à certaines données. Les unités de production sont alors des ressources informatiques placées dans des sites distants et reliés aux consommateurs par des réseaux de transport de données. Ainsi, au lieu de se brancher dans une prise électrique, l'usager se connecterait à un réseau informatique sur lequel il pourrait alors consommer des calculs et/ou des données pour faire fonctionner ses programmes.

L'idée est donc de lier des ressources de calcul et de stockage hétérogènes et distribuées afin que, pour l'utilisateur, elles apparaissent comme une unique entité. Une sorte de puissante machine virtuelle serait alors ainsi formée par le partage de nombreuses machines provenant d'organisations administratives différentes. Le niveau actuel des technologies et le fort intérêt qu'elles suscitent font que les grilles de calcul semblent offrir une solution intéressante aux problèmes du calcul et du stockage intensif.

### 1.2.2 Mise en œuvre

Mettre en place une grille de calcul, c'est faire en sorte que des ordinateurs, de nature hétérogènes, situés dans des sites distants puissent apparaître comme une unique ressource. Au delà de la simple connexion réseau nécessaire pour relier ces machines entre elles il faut rajouter un système dont le rôle est d'assurer la gestion et l'organisation des ressources. Ce système, que l'on nomme « middleware » ou « intergiciel », a pour but d'assurer le lien entre les utilisateurs et les machines, autrement dit, il a pour rôle de répartir les différentes ressources disponibles entre les demandes des usagers.

Il existe de nombreuses façons d'appréhender ce que devrait faire un intergiciel de grille et la façon dont il devrait le faire. Par exemple, le Globus Toolkit [17], l'un des systèmes les plus complets et aboutis actuellement, inclue les logiciels et librairies nécesssaire à la découverte, la gestion et la surveillance des ressources ainsi que la gestion et la sécurité des données. CondorG est un autre système, partiellement basé sur le Globus Toolkit et Condor. Le projet Condor a pour objectif de rendre disponible la puissance de calcul provenant de sources différentes pour l'exécution de tâches. Ces sources peuvent très bien être des machines dédiées, des super calculateurs ou encore des ordinateurs de bureaux inutilisés (la nuit par exemple). Les machines cibles étaient des grappes de PC ou des ordinateurs de bureaux situés au sein d'un même réseau ou domaine administratif. CondorG est la version grille de ce projet permettant

l'utilisation de machines réparties sur des sites distants. Dans les deux cas, il s'agit de systèmes complets mais également très complexes à mettre en oeuvre.

L'approche « Network Enable Server » (NES) est une alternative à ces approches globales. Utilisé dans les intergiciels DIET, NetSolve ou Ninf, il s'agit d'exécuter des tâches sur des serveurs distants dans une approche client-serveur. Généralement plus léger, mais tout aussi complet, les NES sont particulièrement adaptés lorsque les applications à traiter sur la grille sont clairement définies comme la résolution de problème d'algèbre linéaire.

## 1.3 Bioinformatique, calcul et stockage

La figure 1.1 décrit l'architecture classique des applications bioinformatiques. On y retrouve deux grands types de machines connectées par le réseau internet. D'un côté, un ensemble de clients qui soumettent des requêtes à des serveurs de calcul. Les clients sont des ordinateurs individuels qui n'ont aucune connaissances les uns des autres mais qui sont, bien souvent, regroupés sur de même grands sites. Typiquement, il s'agit des ordinateurs de bureaux des chercheurs en biologie ou bioinformatique au sein de centre de recherche. De l'autre coté se trouve les serveurs de calcul qui sont des ordinateurs dédiés au calcul. Il s'agit de puissantes machines, et de plus en plus souvent de grappes de machines. Ces serveurs disposent en local de banques de données et d'algorithmes pouvant s'appliquer sur ces banques. Le nombre de ces banques et algorithmes est bien souvent limité non seulement par l'espace de stockage nécessaire pour les mettre à disposition mais aussi par le coût requis pour les maintenir à jour. En général, ces serveurs sont indépendants les uns des autres et hébergés et maintenus dans les centres de bioinformatiques. Les clients accèdent à ces serveurs via des portails web ou directement en demandant la création de comptes utilisateurs aux administrateurs des serveurs.

Ce modèle d'utilisation et les demandes sans cesse croissantes en temps de calcul et en volume de données à stocker des communautés bioinformatiques et biologiques font des applications bioinformatiques de bonnes candidates à l'utilisation de grille de calcul. Si l'on veut pouvoir utiliser une telle machine de façon efficace et en évitant de gaspiller des ressources, il est indispensable de tenir compte de la profonde nature hétérogène des éléments qui la compose. C'est-à-dire qu'il faut être en mesure de gérer l'espace de stockage distribué ainsi que les différentes ressources de calcul afin d'optimiser l'exécution des requêtes de calcul des utilisateurs. Pour réussir à accomplir cette tâche convenablement, il est indispensable de connaître les caractéristiques, en termes de volume de stockage et de puissance de calcul, des différents composants de la grille ainsi que des informations sur les requêtes d'exécution qui seront soumises à cette grille. Si, en terme général, l'utilisateur est le plus à même de connaître le comportement de son application, il ne peut avoir, où plutôt ne devrait pas avoir, connaissance de la façon dont est constituée la grille qu'il utilise. Ainsi, les tâches d'ordonnancement et de placement des données doivent être effectuées par un système

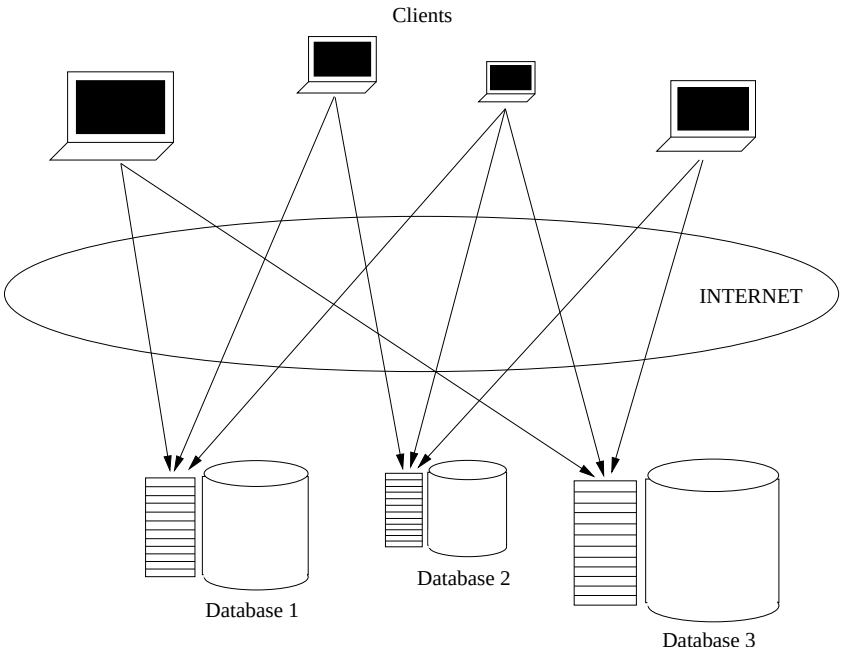


FIG. 1.1 – Vue d'une application bioinformatique.

interne à la grille qui a connaissance des différentes caractéristiques des éléments dont elle est composée. Mais il faudra également à ce système des connaissances sur le fonctionnement des applications afin de répartir, au mieux, les différentes ressources de stockage et de calcul dont la grille dispose. Pour cela, nous nous sommes placé dans un contexte d'applications bioinformatiques particulières qui possèdent des propriétés sur l'exécution et l'utilisation des banques de données bioinformatique qu'il est facile de déterminer.

L'objectif de cette thèse est donc d'établir des mécanismes qui permettront de gérer le placement des données ainsi que l'ordonnancement des requêtes dans un environnement de grille de calcul afin d'optimiser le traitement d'un ensemble de requêtes bioinformatiques.

La thèse comporte cinq chapitres :

- Un état de l'art sur les travaux déjà existant sur l'ordonnancement de tâche et la réplication de données dans un contexte de grille de calcul.
- Une présentation du contexte bioinformatique qui a motivé notre étude ainsi qu'une étude préliminaire sur la gestion des données.
- Présentation du modèle des la plate-forme et des applications puis description d'algorithmes statique et dynamique pour le placement et l'ordonnancement des requêtes.
- Evaluation par simulation des algorithmes précédents.
- Implémentation et évaluation de l'algorithme dans l'environnement de calcul DIET.

Nous passons maintenant en revue ces différents chapitres.

## État de l'art

La gestion des données dans les systèmes à grande échelle a déjà suscité de nombreux travaux de recherche. Il en est de même pour l'ordonnancement des tâches en régime permanent, certaines études, plus récentes ont même conduit à essayer de coupler les deux. Nous présenterons dans ce chapitre un état de l'art sur les différents domaines qui sont en relation avec la gestion et la réplication des données ainsi que le couplage qui peut être fait entre l'ordonnancement des calculs et la réplication des données.

## Problématiques

Dans ce chapitre, nous allons présenter en détail les motivations et la problématique que nous avons cherchée à résoudre. Nous effectuerons une présentation générale du cadre des applications bioinformatiques auxquelles nous nous sommes intéressé. Nous montrerons ensuite, grâce à des traces réelles d'exécution d'un serveur de calcul bioinformatique, que cette catégorie d'applications possède des caractéristiques bien particulières dans l'utilisation des données. Ces constatations nous conduirons à nous intéresser à la gestion des données bioinformatiques. Tout d'abord, une première étude de gestion des données en utilisant des techniques primaires issues de la gestion des caches nous montrera que pour obtenir des performances régulières, il semble indispensable de coupler gestion des données et ordonnancement des requêtes de façon très étroite.

## Algorithmes

Gérer de façon simultanée le placement des données et l'ordonnancement des tâches sur ces données devrait permettre une gestion plus efficace de l'espace de stockage ainsi que de la puissance de calcul disponible sur la plate-forme. Dans cette partie, nous allons présenter les algorithmes que nous avons élaborés afin de répondre à cette problématique. Pour cela, il est nécessaire de définir clairement les différents paramètres de la plate-forme ainsi que des différentes informations régissant l'utilisation des données. À partir de l'analyse précédente basée sur des traces réelles d'exécution d'un portail bioinformatique, nous exposerons une modélisation des requêtes et de leur comportement. Nous avons alors établi un algorithme statique basé sur la résolution d'un programme linéaire décrivant les différentes contraintes liées au placement des données et à l'exécution des requêtes. Mais, si la grille de calcul est un milieu fortement hétérogène d'un point de vue du matériel qui la constitue, elle l'est aussi dans

les usagers qui s'en servent et l'utilisation qu'ils en font. Ainsi, afin d'être capable d'adapter notre placement et notre ordonnancement en cas de de variations dans l'utilisation des données, nous avons défini des heuristiques permettant au système de gestion d'adapter dynamiquement placement des données et ordonnancement dans de tel cas. Ces heuristiques se basent sur la surveillance des serveurs de calcul afin de repérer les anomalies au cours de l'exécution d'un flot de requêtes pour en déterminer l'origine et proposer des solutions pour optimiser le fonctionnement de la plate-forme.

## Évaluations

Nous avons cherché à évaluer l'efficacité de nos algorithmes et heuristiques. Pour cela, nous avons choisi d'effectuer, dans un premier temps, des simulations avec le simulateur de grille OptorSim dédié à la gestion des données dans les grilles de calcul. Nous commencerons par expliciter le fonctionnement de ce simulateur. Nous détaillerons ensuite l'ensemble des modifications que nous avons dues lui apporter, que ce soit pour pouvoir intégrer tous les paramètres de notre modèle ou implémenter nos algorithmes et heuristiques. Nous verrons alors les performances obtenues par notre méthode statique comparé à des heuristiques gloutonnes de placement et des algorithmes d'ordonnancement dynamique. Nous montrerons enfin les résultats obtenus par nos différentes heuristiques d'adaptation dynamique du placement et de l'ordonnancement lorsque l'utilisation réelle des données diffère des informations connues lors du placement initial.

## Implémentation

Enfin, nous avons voulu vérifier qu'il était possible de mettre nos algorithmes en œuvre dans un intergiciel de grille existant et de vérifier que cela permettait d'améliorer les performances de la plate-forme. Pour cela, nous avons implémenté une partie de nos méthodes au sein de l'environnement logiciel de grille *Distributed Interactive Engineering Toolbox (DIET)* et du gestionnaire de données qui lui est associé : *Data Tree Manager (DTM)*. Nous commencerons par présenter l'architecture et le fonctionnement de ces deux outils. Nous verrons ensuite comment il nous a été possible d'intégrer nos mécanismes au sein de cette architecture. Enfin, nous avons déployé l'outil et effectué des tests de comparaisons entre notre méthode et les mécanismes natifs d'ordonnancement et de placement du couple DIET-DTM.

# Chapitre 2

# État de l'art

## 2.1 Introduction

La gestion de données à grande échelle n'est ni un problème récent, ni un problème spécifique aux grilles ! Les travaux de recherche autour d'une gestion efficace de ces données sont très nombreux et couvrent des domaines d'applications très variés. On trouve par exemple des travaux autour des bases de données distribuées, autour du Web et de ses caches, des grosses applications de simulations ou d'imagerie à grande échelle.

Les grilles de données ont cependant des caractéristiques qui les distinguent des autres systèmes. Nos systèmes et applications cibles manipulent des données de très grandes tailles et les travaux effectués sur ces données ont un coût non négligeable. La réplication a donc un rôle primordial pour l'obtention de bonnes performances. Un excellent état de l'art des grilles de données en général est donné dans [41]. Les fonctionnalités essentielles de ce type d'infrastructure sont des protocoles de communication à haute performance et sécurisés pour le transfert rapide de grandes quantités de données et un mécanisme de réplication extensible qui permettra d'avoir une distribution des données à la demande. Tout ceci doit bien évidement être fait dans une transparence la plus importante possible pour éviter à l'utilisateur d'avoir à choisir telle ou telle source de données. Un bon exemple d'application manipulant de grandes quantités de données à grande échelle est certainement l'application LHCb du CERN qui rentrera en production dès 2007. Des péta-octets de données seront générés chaque année et des physiciens du monde entier devront y avoir accès.

## 2.2 Gestion de données pour le Web et CDN

Le Web constitue un des domaines de recherche les plus actifs autour de la gestion de données à grande échelle. Les sources de données sont multiples et réparties

géographiquement, les utilisateurs très nombreux et le nombre de requêtes astronomique. La réplication et la gestion "intelligente" de l'ordonnancement des requêtes de demandes de document sont donc obligatoires pour absorber la charge.

Les techniques de gestion des caches pour le web est un domaine, toujours actif, pour lequel un très grand nombre de travaux et de techniques ont déjà été developpé. Dans [42], l'auteur passe en revue l'ensemble des endroits où l'optimisation du traitement des requêtes et de la gestion des données peut avoir lieu. Il expose également les grandes techniques pour effectuer ces réalisations.

L'article [33] présente également les techniques mises en œuvre dans les caches web, mais cette fois, les auteurs se sont attachés au point particulier des techniques de gestion des données dans les caches. Ils classifient ces différentes stratégies en cinq grandes familles :

- stratégies basées sur l'ancienneté
- stratégies basées sur la fréquence d'accès
- stratégies basées sur un ratio entre ancienneté et fréquence
- stratégies basées sur une fonction
- stratégies aléatoires

L'ancienneté et les fréquences d'accès aux données du cache sont deux facteurs très importants mais également très différents. Les représentants les plus connus sont les méthodes LRU (Last Recently Used) et LFU (Last Frequently Used). Ces deux méthodes peuvent être déclinées et combinées pour prendre en compte d'autres paramètres comme un seuil qui déterminera des données qui resteront fixes dans le cache, ou encore en cherchant à minimiser le nombre d'éléments qui seront déplacés. Les stratégies basées sur des fonctions évaluent une fonction spécifique prenant en compte différents facteurs qui peuvent être, évidemment, l'ancienneté ou la fréquence d'accès, mais aussi la taille des données ou une connaissance de la distribution des requêtes. Enfin, les stratégies utilisant un générateur aléatoire offre une approche différente et non déterministe généralement très simple à mettre en œuvre.

Dans un domaine connexe, les « *Content Distribution Networks* » (CDN) proposent des solutions pour offrir à l'utilisateur des serveurs annexes pour limiter la charge d'un serveur principal [32]. Les requêtes demandant des documents sont donc redirigées suivant une heuristique d'équilibrage de charge. Cela est effectué grâce à une modification du DNS[1]. Si la donnée n'est pas/plus présente, la requête est encore une fois redirigée. Les problématiques derrière les CDN consistent donc à la fois à choisir les bons serveurs où disposer les données répliquées et à rediriger les requêtes de manière équilibrée.

[1] Domain Name Server

## 2.3 Systèmes pair-à-pair

La gestion et la réplication de données sont également au centre des systèmes pair-à-pair. La réplication est souvent effectuée par les utilisateurs eux-mêmes comme par exemple avec le système Gnutella [24]. Le système Freenet [13] quand à lui a une stratégie de réplication automatique lorsqu'un fichier devient très populaire. Cette réplication est effectuée de manière cryptée chez des utilisateurs du système en utilisant une stratégie de type *Last Recently Used* (LRU). Ces systèmes sont avant tout orientés vers le partage de données et non le stockage en vue d'une utilisation pour des calculs.

Il existe d'autres projets comme OceanStore [25], PAST [38] ou US [39] dont l'objectif est centré sur le stockage des données à long terme. Les systèmes pair-à-pair sont souvent fortement dynamiques et ces projets proposent des solutions permettant la disparition d'un ou plusieurs des nœuds de stockage tout en conservant la disponibilité de l'ensemble des données stockées. Dans ce cas, le stockage plus que le partage est au centre du problème.

Cependant, ces systèmes diffèrent des grilles traitées dans cette thèse car il s'agit avant tout de partage de données à grande échelle. Les tailles de stockage (et les données manipulées) sont très inférieures à notre problème et enfin, aucun calcul n'est effectué sur les nœuds possédant ces données. Il s'agit donc souvent d'une approche plus « système » de la réplication.

## 2.4 Réplication dans les grilles de données

### 2.4.1 Algorithmes et heuristiques de réplication

De nombreux travaux concernent l'ordonnancement de tâches indépendantes sur les plates-formes hétérogènes et distribuées. La complexité de la réplication dans les grilles de données a été présentée dans [12]. Les auteurs montrent que le problème est NP-complet et non-approximable et donnent deux approches pour le résoudre grâce à une simplification du problème.

Dans [34], les auteurs comparent plusieurs stratégies dynamiques de réplication dans un environnement de grille hiérarchique. Les stratégies sont comparées en mesurant (par simulation) le temps de réponse moyen et la bande-passante totale utilisée. Les deux stratégies qui obtiennent les meilleurs résultats sont *cascade* qui inonde l'arbre modélisant la grille un peu comme une fontaine en utilisant des mesures des besoins à chaque niveau. Cette stratégie fonctionne bien dans le cas où il n'y a pas beaucoup de localité. La seconde stratégie qui a obtenu de bons résultats dans le cas de requêtes aléatoires est *diffusion rapide* dans laquelle une donnée copiée à chaque niveau sur le chemin vers le client.

Dans [2], un modèle économique de gestion (création et destruction) de réplicats est proposé et évalué expérimentalement dans [3]. Dans ce système, un agent est situé

sur chaque nœud de stockage et se sert d'un protocole d'« enchères » pour choisir quel réplicat d'un fichier est à utiliser. Lorsqu'une donnée est requise sur un site, l'agent concerné va interroger les serveurs de stockage. Le serveur qui remporte l'enchère est celui qui a proposé le prix le plus faible et il se voit alors rétribué du prix de la seconde enchère la moins élevée. Pour chaque serveur interrogé, si la donnée est présente alors le prix fixé est proportionnel au temps estimé pour le transfert de fichier entre le serveur de stockage considéré et le site demandeur. Si la donnée n'est pas présente, le serveur de stockage a la possibilité de déclencher lui aussi une demande d'enchère pour acquérir la donnée s'il estime que les revenus qu'elle va lui apporter seront plus grands que le coût de son achat. Une enchère initiale peut donc engendrer des enchères en cascade. Bien évidemment, cela suppose que les serveurs de stockage ont un moyen de prédiction de l'utilisation des données pour pouvoir estimer les revenues qu'elles peuvent générer.

Dans [43], les auteurs proposent ADR (Adaptive Data Replication) un algorithme qui adapte la réplication des données dynamiquement en fonction des accès qui sont fait à celles-ci. Le but de l'algorithme est d'optimiser les accès distants, en lecture et en écriture, à des données en les répliquant de façon à être au plus près de l'endroit où elles sont utiles tout en évitant les copies superflues. Dans le cas où plusieurs réplicats d'une même donnée existes, la lecture peut se faire sur n'importe quel réplicat, par contre, un accès en écriture devra être propagé à tous les réplicats. À intervalle régulier, tous les nœuds stockant des données vérifient les accès en lecture et en écriture qui ont été faits sur les données qu'ils stockent. En fonction de cela, ils déterminent s'ils doivent propager leurs données aux nœuds qui l'entourent ( par exemple s'il y a plus d'acces en lecture provenant de nœuds distants), s'ils suppriment des données (par exemple avec un grand nombre d'écritures et peu de lectures) ou s'ils échangent des données avec un serveur voisin. Les auteurs ont prouvé que si les schémas d'accès en lecture et écriture sont réguliers, alors leur algorithme permet de converger vers une réplication optimale.

D'autres types d'algorithmes prennent en compte la localité des données. Dans [31], les serveurs sont regroupés en différentes régions en suivant la topologie du réseau. Les communications entre les nœuds d'une même région devraient être assez rapide. Lorsque une donnée est nécessaire sur un serveur et qu'il n'y a plus de place pour la stocker alors l'algorithme BHR (Bandwidth Hierarchy based Replication) proposé cherchera à récupérer la donnée en question uniquement si elle n'est pas déjà présente sur un des nœuds de la même région. Si la donnée n'est pas présente, alors de la place est faite sur le serveur en supprimant les données les plus anciennement accédées et qui sont présentes ailleurs dans la région.

## 2.5 Couplage ordonnancement réplication

L'approche défendue dans la thèse concerne le couplage fort de la réplication et de l'ordonnancement des requêtes pour lequel peu de travaux existent.

Plusieurs stratégies de réplication et ordonnancement combinées sont étudiées dans [35]. Les auteurs proposent un modèle à deux niveaux constitué de différents sites possédant chacun des ressources de calcul, de stockage et des utilisateurs qui lancent des tâches de calcul utilisant les données répliquées. Sur chaque site on trouve également un ordonnanceur local qui détermine l'ordre dans lequel les tâches sont exécutées une fois qu'elles ont été affectées à ce serveur et un gestionnaire de données dont le rôle consiste à choisir le bon moment pour répliquer les données et/ou détruire des données locales pour gagner de la place de stockage. En plus de ces deux entités d'ordonnancement locales, on trouve des ordonnanceurs externes chargés de répartir les requêtes entre les sites. Plusieurs combinaisons de stratégies d'ordonnancement et de réplication sont ensuite comparées.

Dans [11], les auteurs étudient l'ordonnancement de requêtes de calcul et la gestion de la réplication. Ils proposent l'algorithme IRS (Integrated Replication and Scheduling) dans lequel les performances sont itérativement améliorées en affinant successivement le placement des données et l'ordonnancement des tâches de calcul. Dans ce cas, l'ordonnancement est calculé en fonction des informations fournies par le système de réplication et les réplications sont faites en fonction d'informations fournies par l'ordonnanceur. L'ordonnancement est fait à la soumission d'une requête et deux approches sont proposées. La première se base sur la disponibilité des données : si une requête nécessite plusieurs données alors elle sera envoyée sur le site qui en dispose le plus grand nombre, en cas d'égalité entre deux serveurs l'estimation du temps nécessaire pour transférer les données manquantes est utilisée pour les départager. La deuxième approche se fait en fonction du coût de l'ordonnancement. Ce coût est calculé pour chaque site en fonction du temps requis pour le transfert des données nécessaire, le temps d'attente dans la file du serveur (les requêtes sont exécutées l'une après l'autre) et le temps d'exécution de la requête elle même sur le serveur. La requête est alors envoyé sur le site qui obtient le coût le plus faible. À chaque soumission de requête, une matrice faisant étant des demandes des données sur chaque site est mise-à-jour. Celle-ci servira de base pour l'algorithme de réplication qui est mis en œuvre périodiquement suivant un intervalle de temps fixe. Cet algorithme se déroule en deux phases. La première consiste à définir un seuil du nombre de réplications maximales d'une donnée, les données les plus utilisées sont celles qui pourront être le plus répliquées. La deuxième phase est la réplication. Son objectif est de minimiser globalement le temps d'accès aux données. Cela est réalisé à partir de la matrice d'utilisation des données sur les différents serveurs.

# Chapitre 3

# Etudes préliminaires

## 3.1 Motivations

Ce travail est motivé par une application en Sciences de la Vie dans le domaine de la recherche de sites et signatures de protéines dans des banques de données de séquences de protéines [19]. Les programmes d'acquisition en génomique, comme les projets de séquençage de génomes complets, produisent un volume de données de plus en plus important généralement mis à disposition de l'ensemble des membres des communautés biologiques et bioinformatiques. Mais il s'agit là de données brutes qui doivent être comprises et annotés afin d'être réellement utiles pour d'autres études. Il existe un grand nombre d'outils et d'algorithmes provenant des différents domaines de la bioinformatique (similarité et homologie, analyse de séquences, analyse des fonctions des protéines, etc.) dont le but est de permettre l'analyse et le traitement de ces données. Les sites fonctionnels et les signatures de protéines sont très utiles pour analyser ces données ou pour corréler différentes sortes de données biologiques existantes. Les sites et signatures de protéines peuvent être exprimés en utilisant la syntaxe définie par la banque de données PROSITE [6] écrite sous la forme d'une expression régulière sur l'alphabet des acides aminés. La recherche de tels sites ou signatures dans des banques de données se rapproche alors fortement de la recherche de motifs. Toutefois, il ne s'agit pas tout à fait de la même chose puisque dans notre cas, des erreurs dans le motif ou la séquence recherché peuvent être tolérées si elles ont une signification biologique. Par exemple, ces méthodes peuvent être utilisées pour identifier et essayer de trouver une caractérisation des fonctions potentielles d'une nouvelle protéine, ou pour regrouper en famille de protéines des séquences contenues dans des banques internationales. Dans la plupart des cas, pour ce type d'analyse, le temps d'exécution est relativement court et dépend principalement de la taille des banques de données. Mais si le temps d'exécution d'une requête est court, le nombre de requêtes est très important.

Les difficultés pour la gestion de tels ensembles de données proviennent du fait que le nombre de banques de données utilisé pour ce type de recherche peut être

très important, tout comme les écarts entre les tailles de ces différentes banques. Ces ensembles de données peuvent être des banques de séquences de protéines internationales telles que Swiss-Prot/TrEMBL [4], PIR [44], etc. Ces données peuvent également venir directement de projets de sequencage de génome, dans ce cas le nombre d'ensembles de données est aussi important que la taille du génome.

## 3.2 Hypothèses générales

### 3.2.1 Les banques de données et leur utilisation

Les deux points cruciaux sur lesquels reposent notre étude sont les banques de données et les requêtes qui sont effectuées dessus. Il nous fallait pouvoir faire des hypothèses fortes et réalistes afin de construire un système optimisé au mieux. Nous avons eu l'avantage d'avoir accès aux traces d'exécution du serveur NPS@ [30, 14] hébergé et géré par l'équipe de BioInformatique de L'Institut de Biologie et Chimie des protéines de Lyon [22]. NPS@, Network Protein Sequence Analysis, est un serveur dédié à l'analyse de séquences de protéines. Il est mis à la disposition de toute la communauté biologique par le biais d'un portail web accessible à l'adresse `http://npsa-pbil.ibcp.fr`. L'exécution des requêtes soumises via une interface web [30] se fait sur une grappe de 36 processeurs sous linux avec un espace de stockage global de $1,5To$. Ce serveur offre aux utilisateurs de la communauté bioinformatique l'accès à un certain nombre de banques de données biologiques de référence ainsi que la possibilité d'exécuter sur ces banques de données un certain nombre d'algorithmes reconnus dans la communauté d'études des protéines. Au cours de l'année 2004, presque deux millions de requêtes ont été traitées par ce serveur. Bien qu'offrant une bonne capacité de calcul et de stockage, cette grappe ne permet pas d'offrir à ses utilisateurs l'ensemble des banques de données et des algorithmes disponibles, seul un sous ensemble de ceux-ci peut être mis a disposition. Cependant, ce sous ensemble est suffisamment large pour que les déductions que nous avons pu tirer de leur utilisations soient considérées comme caractéristiques des applications auxquelles nous nous intéressons.

Les traces auxquelles nous avons eu accès, dont nous pouvons voir quelques lignes dans la figure 3.1, contiennent pour chaque requête, le nom de l'algorithme utilisé, le nom de la banque de données de référence, ainsi que les dates de début et de fin de l'exécution de la requête. La table 3.1 montre quelques informations extraites de ces traces d'exécution.

En étudiant ces traces, nous avons pu faire plusieurs constatations très importantes pour la suite. La première constatation concerne le schéma global d'accès au données. Si nous prenons deux échantillons suffisamment larges de traces couvrant des périodes de temps équivalentes, nous pouvons constater, qu'à de petites variations près les fréquences d'accès aux banques de données et aux algorithmes restent constantes.

```
02/24/04 17:17:53;02/24/04 17:17:56;blastp;/db/TrEMBL/sp.fas;3;
02/24/04 17:19:13;02/24/04 17:19:16;blastp;/db/TrEMBL/sp.fas;1;
02/24/04 17:19:40;02/24/04 17:19:46;blastp;/db/TrEMBL/sp.fas;1;
02/24/04 17:20:42;02/24/04 17:20:51;pattinprot;/db/TrEMBL/sp.fas;1;
02/24/04 17:21:22;02/24/04 17:21:25;blastp;/db/TrEMBL/sp.fas;1;
02/24/04 17:23:13;02/24/04 17:23:16;blastp;/db/TrEMBL/sp.fas;1;
02/24/04 17:24:05;02/24/04 17:24:08;blastp;/db/TrEMBL/sp.fas;1;
02/24/04 17:35:52;02/24/04 17:38:16;psi-blast;/db/Nr_Prot/nr.fas;1;
02/24/04 17:40:54;02/24/04 17:44:42;psi-blast;/db/TrEMBL/sp.fas;1;
02/24/04 17:48:05;02/24/04 17:48:44;blastp;/db/TrEMBL/tr.fas;1;
```

FIG. 3.1 – Extrait des traces d'exécutions de serveur bioinformatique NPS@. La première date est la date de début de l'éxécution de la requête, la deuxième est celle de fin, vient ensuite le nom de l'algorithme utilisé ainsi que le chemin d'accès à la banque de données. La dernière valeur est un code définissant les droits de l'utilisateur.

| | |
|---:|:---:|
| Nombre de banques de données | 23 |
| Nombre d'algorithmes | 8 |
| Nombre de couple algorithmes-banques | 80 |
| Nombre de requêtes | 88730 |
| Taille moyenne des banques | 1 Go |
| Taille de la plus petite banque | 1 Mo |
| Taille de la plus grande banques | 12 Go |

TAB. 3.1 – Informations extraites des traces d'exécutions

C'est-à-dire que les requêtes effectuées sur deux intervalles de temps sont pratiquement identiques. Ainsi, connaître les fréquences d'accès et d'utilisation d'une période permet d'avoir une bonne évaluation des accès qui vont se produire. Pour confirmer ce résultat, nous avons extrait $750$ ensembles de $40000$ requêtes successives des traces d'exécution. Pour chacun des ensembles, nous avons alors déterminé le pourcentage de requêtes dans lequel chacun des algorithmes était impliqué. La figure 3.2 représente la répartition de ces pourcentages d'apparitions pour quatre des algorithmes présents. Nous y voyons, par exemple, que dans presque $65\%$ des ensembles l'algorithme *pattinprot* est présent dans $32\%$ des requêtes. Cependant une valeur de $65\%$ n'est pas suffisante pour pouvoir généraliser sur le fait que l'algorithme *pattinprot* représente $32\%$ des requêtes. Nous avons donc cherché à regarder quelle répartition nous avions sur un voisinage de points. C'est-à-dire qu'au lieu de regarder dans quel pourcentage des échantillons les fréquences d'apparition d'un algorithme étaient obtenues, nous avons considéré les fréquences à plus ou moins deux pourcent. Pour reprendre l'exemple précédent, cela correspond à regarder dans quel pourcentage des échantillons, la fréquence d'apparition de l'algorithme *pattinprot* est comprise entre $30\%$ et $34\%$. C'est ce que nous pouvons voir sur la figure 3.3 réalisée à partir des mêmes données que la fi-

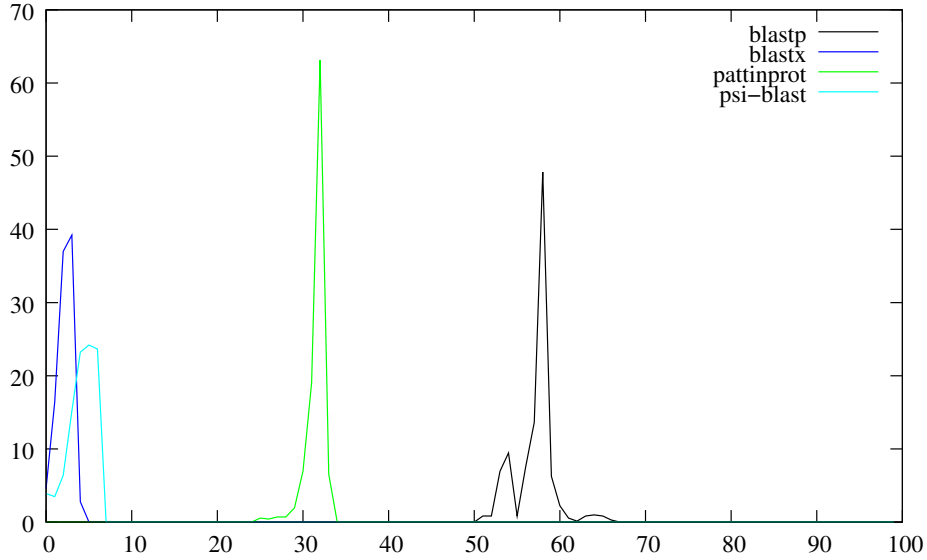


FIG. 3.2 – Répartition des fréquences d'apparitions.

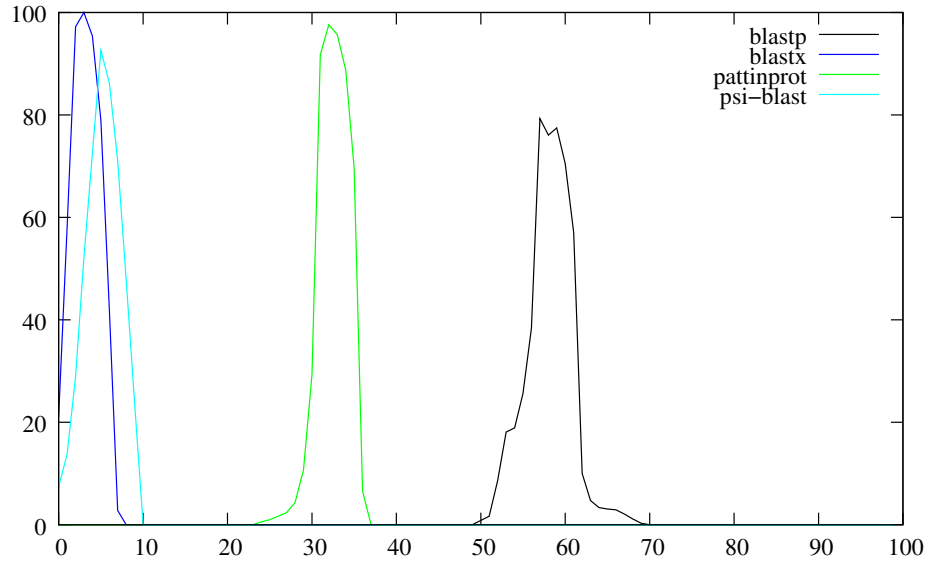


FIG. 3.3 – Répartition cumulée sur $5\%$ des fréquences d'apparitions.

gure 3.2. Nous constatons alors que dans presque $100\%$ des cas, l'algorithme *pattinprot* représente entre $30\%$ et $34\%$ des requêtes. Pour les trois autres algorithmes représentés sur cette figure nous obtenons une fréquence d'apparition présente dans plus de $80\%$ des cas. Ainsi, notre hypothèse sur la stabilité des fréquences d'apparition est confirmée.

La deuxième hypothèse est encore inspirée des traces d'exécution dont nous disposons. Les temps de calcul laisse suggérer que le temps d'exécution des algorithmes est

affine en la taille de la banque de données sur laquelle ils sont exécutés. Les traces dont nous disposons permettent de penser que cette hypothèse est réaliste mais le nombre de valeurs différentes, c'est-à-dire d'exécution avec des données de tailles différentes, est un peu trop restreint pour pouvoir l'affirmer aussi complètement que l'hypothèse de stabilité des fréquences d'apparition de chaque banque. Cependant, dans [27], Alan Su montre cette propriété sur l'algorithme *pattinprot* en l'exécutant avec un nombre croissant de motif de recherche. Ce qui nous conforte dans notre hypothèse sur un temps de calcul affine en la taille des données pour la catégorie d'algorithmes que nous considérons.

## 3.3 La gestion du cache

### 3.3.1 Introduction

Dans le chapitre précédent, nous avons vu que la recherche sur les caches web a conduit à de nombreux algorithmes permettant une gestion efficace des données présentes sur des serveurs de données. Parmi ces méthodes, deux facteurs importants reviennent fréquemment. Ces facteurs sont l'ancienneté des accès des données présentes sur le serveur ainsi que leur fréquence d'accès. Nous avons donc cherchés à voir si des techniques simples, mais qui ont prouvé leur efficacité, basées sur ces paramètres pouvaient donner de bons résultats dans notre cas de figure. Dans un premier temps, nous présenterons donc deux algorithmes connus pour la gestion des caches. Nous proposerons ensuite des variantes à ces techniques prenant en compte la taille des données. Enfin, à l'aide d'un simulateur, nous testerons l'efficacité de ces méthodes.

### 3.3.2 Rappels

Dans un premier temps, nous avons donc décidé de voir si des algorithmes simples qui donnent des résultats satisfaisants dans le cas de cache web, ou de gestion de mémoire cache, pouvaient s'appliquer dans notre cas. Nous avons donc choisi de considérer les espaces de stockage offerts par les nœuds de calcul comme des caches de données dont on gérerait le contenu avec une politique fixe et bien définie. Le principe de base d'un cache est assez simple, il s'agit d'un espace de stockage dans lequel sont stockées les données qui sont requises sur les serveurs. Ainsi à chaque requête qui va doit être exécuter, si la donnée n'est pas présente dans le cache, il faut la rapatrier depuis un autre serveur. Dans ce cas, deux possibilités : soit le cache a suffisamment d'espace libre pour stocker cette donnée, alors la donnée peut être stocké sans problème, soit l'espace disponible n'est pas assez important, il faut alors choisir une ou plusieurs données déjà stockées et les retirer du cache afin de libérer de l'espace pour la nouvelle donnée. La façon dont sont choisies les données qui doivent être vidées du cache se fait suivant une politique de choix.

Les deux techniques de gestion de cache les plus simples et aussi les plus répandues sont les politiques de gestion *Last Frequently Used (LFU)* et *Last Recently Used (LRU)*. LFU consiste à considérer la fréquence d'accès aux données. L'idée qui se cache derrière cette heuristique est qu'une donnée fréquemment accédée dans le passé a de fortes chances de l'être toujours autant dans le futur. Ainsi, il serait plus rentable de conserver les données les plus utilisées. Le serveur de stockage maintient une liste des nombres d'accès pour chacune des données. Lorsqu'il faut choisir une donnée à retirer du cache, nous choisissons alors celle qui à été la moins accédée parmi l'ensemble des données présentes.

La politique LRU, quant à elle, ne regarde pas les fréquences d'accès aux données mais la date du dernier accès, partant du principe qu'une donnée qui n'a pas été utilisé depuis longtemps à de grandes chances de ne pas être réutilisée avant longtemps. Ainsi, pour mettre LRU en œuvre, il faut, pour chaque donnée dans le cache, conserver la date du dernier accès à la donnée. Lorsqu'une suppression est nécessaire, la donnée candidate à la suppression est alors celle dont la date d'accès est la plus ancienne dans l'ensemble du cache.

Ces deux stratégies sont simples mais généralement assez efficaces, leurs postulats de bases étant souvent vérifiés. Mais dans notre situation, ces deux stratégies ne prennent pas suffisamment le reste de l'environnement en considération. Le principal exemple est la taille des données. Ces deux politiques ont été conçues à l'origine pour la gestion des caches mémoire de processeurs afin d'accélérer les traitements. Cela implique qu'elles travaillaient sur des ensembles de blocs de données de taille fixe. Lorsqu'elles ont été appliqués pour les caches web, si les données n'étaient plus exactement de même taille, elles restaient de petites tailles (une page web) et généralement toutes du même ordre de grandeur. Dans notre cas, la taille des données peut être très importante et les écarts de tailles entre deux données distinctes être très grands.

Nous avons donc modifié les heuristiques de départ afin d'essayer de prendre en compte ces deux différences.

### 3.3.3 LRU, LFU et taille des données

Les principes de base de LFU et LRU sont relativement cohérents pour les environnements que nous ciblons. L'utilisation des données, qu'on la prenne en compte par la fréquence d'accès ou la date de dernier accès, est une variable importante pour déterminer l'utilité de la présence d'une donnée dans l'espace de stockage. Mais dans le même temps, cette variable n'est pas la seule à prendre en compte. En effet, les données que nous considèrons ont des tailles très diverses et peuvent atteindre des volumes très importants (plusieurs Go pour les plus grandes). Dans de telles conditions, le transfert, comme le stockage, de ces données a un coût qui peut alors parfois s'avérer très élevé. D'un coté, conserver une donnée de grande taille sur un serveur, meme si son utilisation n'est pas la plus importante peut permettre d'éviter les coûts du transfert de cette donnée si elle venait à être requise à nouveau sur ce serveur. Et

dans un environnement de grille utilisant, par exemple, le réseau internet, le transfert de données de grandes tailles peut prendre un temps très important. Mais, si le transfert est coûteux, l'espace utilisé ne l'est pas forcement moins. En effet, une donnée volumineuse pourrait occuper la place de plusieurs données aux tailles plus reduites. Conserver la donnée importante alors que son utilisation est inférieure aux autres entraîne alors que de petites données soient supprimées puis rapatriées à nouveau à de multiples reprises. Ces mouvemments de données de petites tailles pourraient être éviter en commençant par nettoyer le cache des données volumineuses les moins utilisées.

Nous avons alors établi quatre nouvelles heuristiques basées sur LRU ou LFU et prenant en compte également la taille des données.

**LfuBig** : LFU et grandes données.
La première heuristique est basée sur LFU. Lorsqu'il est nécessaire de supprimer nous commençons par sélectionner un ensemble de donnée en se basant sur le critère de la fréquence d'accès. Toutes les données dont le nombre d'accès est inférieur à un seuil fixé comme un pourcentage de la fréquence d'utilisation la plus forte sont sélectionnable. Si aucune des données n'est inferieure au seuil, ce dernier est progressivement augmenté jusqu'à ce qu'il permette de sélectionner au moins deux données. Ensuite, parmi les données retenues, la donnée à supprimer sera celle qui aura la plus grande taille.

**LfuSmall** : LFU et petites données.
Nous avons également défini *LfuSmall* dont la seule différence avec *LfuBig* est que la donnée sélectionné parmi l'ensemble retenu sera la donnée de plus petite taille.

**LruBig** : LRU et grandes données.
Le principe reste le même que pour les deux heuristiques précédentes à savoir qu'un premier ensemble de données pouvant être supprimées est déterminé, tandis que le choix final se fait selon le critère de la taille. Dans ce cas, la première sélection des données est basée sur la date de dernier accès, c'est-à-dire suivant le principe de LRU. Toutes les données qui n'ont pas été accédées depuis une date seuil sont potentiellement supprimables. Le choix se fait ensuite parmi ces données selon le critère de la taille : la donnée dont la taille est la plus grande sera supprimée. La date seuil est défini par rapport aux dates d'accés de la donnée la plus récemment accédée et la plus anciennement accédee.

**LruSmall** : LRU et petites données.
La différence avec *LruBig*, consiste dans le choix final de la donnée à supprimer. Dans cette méthode, c'est la donnée la plus petite qui sera éliminée.

Nous avons donc quatres nouvelles heuristiques :

**LfuBig** sélection d'un ensemble de données basées sur LFU, puis choix de la donnée la plus grande.

**LfuSmall** sélection d'un ensemble de données basées sur LFU, puis choix de la donnée la plus petite.

**LruBig** sélection d'un ensemble de données basées sur LRU, puis choix de la donnée la plus grande.

**LruSmall** sélection d'un ensemble de données basées sur LRU, puis choix de la donnée la plus petite.

### 3.3.4 Simulations et résultats

Nous avons testé ces différentes heuristiques par des simulations. Le simulateur utilisé est une version d'OptorSim que nous avons largement modifiée afin de tester l'ensemble de nos algorithmes et heuristiques. L'ensemble des détails concernant le choix, le fonctionnement et les modifications du simulateur seront apportés plus tard dans la section 5.2, une fois que toutes les hypothèses de travail auront été posées. Sans rentrer dans les détails, il suffit de savoir qu'OptorSim est un simulateur de grille de calcul dédié à la gestion des données. Nous avons implémenté nos quatres heuristiques dans le simulateur afin de voir l'impact de la gestion du cache sur le temps de calcul. À partir de l'étude des traces du serveur NPS@ décrit précédemment, nous avons généré des ensembles de requêtes ayant les mêmes caractéristiques. Dix ensembles de 20000 requêtes ont été ainsi générés, tous respectant la même fréquence d'apparition des types de requêtes.

Nous avons effectué ces simulations pour les quatre heuristiques que nous avons établies ainsi que pour les stratégies LRU et LFU de base dans un but de comparaison. Le tableau 3.2 montre les temps d'exécution pour les différentes heuristiques de gestion de caches. Le tableau 3.3 donne des chiffres concernant le transfert des données au cours des simulations. Les valeurs de ces tableaux sont des moyennes sur les dix tests effectués pour chacunes des heuristiques. Si l'on observe ces valeurs, la première constatation est qu'aucune des stratégies ne se distingue réellement des autres. D'un point de vue du temps d'exécution, les méthodes LfuBig et LruBig sont, en moyenne, les moins efficaces. Si l'on regarde d'un point de vue des données transfert, on retrouve le même résultat, puisque ces deux stratégies sont celles qui, globalement, font le plus de transfert, aussi bien en nombre de transfert qu'en volume de données et qu'en temps nécessaire pour effectuer ces transferts. Ces deux méthodes privilégient la suppression des données de grandes tailles lorsqu'il est nécessaire de supprimer des données dans l'espace de stockage. Or, si l'on observe les schémas d'utilisation des données, les banques de données de plus grandes tailles sont celles qui sont les moins utilisées, ce qui explique qu'elles sont souvent supprimées. Cependant, elles sont tout de même régulièrement utilisées et ne peuvent se contenter d'être stockées sur un unique serveur. Les phases de suppression - réplication sont donc fréquentes et très coûteuse. La deuxième constatation importante concerne les temps minimum que l'on peut observer. Nous voyons que pour certaines simulations le temps d'exécu-

| | temps moyen | min | max |
|---|---|---|---|
| LFU | 41982 s | 27762 s | 64435 s |
| LRU | 42479 s | 24943 s | 65702 s |
| LfuSmall | 40943 s | 33498 s | 68554 s |
| LruSmall | 43400 s | 35775 s | 68281 s |
| LfuBig | 46313 s | 35212 s | 75595 s |
| LruBig | 46694 s | 37318 s | 62029 s |

TAB. 3.2 – Temps d'éxécution de 20000 requêtes avec un ordonnancement de type MCT pour différentes stratégies de gestion de caches basées sur LRU et LFU (valeurs moyennes sur dix simulations).

| Heuristiques | volume transféré | nb de transfert | temps de transfert cumulés |
|---|---|---|---|
| LFU | 284246 Mo | 872 | 1398 s |
| LRU | 285794 Mo | 897 | 1427 s |
| LfuSmall | 283427 Mo | 863 | 1425 s |
| LruSmall | 224664 Mo | 748 | 1327 s |
| LfuBig | 247569 Mo | 757 | 1551 s |
| LruBig | 261119 Mo | 778 | 1747 s |

TAB. 3.3 – Information sur les transferts de données effectuées durant l'éxécution de 20000 requêtes avec un ordonnancement de type MCT pour différentes stratégies de gestion de caches basées sur LRU et LFU (valeurs moyennes dix simulations).

tion peut être très inférieur au temps moyen observé sur l'ensemble des simulations. Les écarts sont tellement importants que nous avons tout d'abord pensé à un problème dans le simulateur qui aurait pu causer des mesures erronées. Mais une étude détaillée des traces générées par le simulateur au cours des expériences nous a permis d'exclure cette possibilité. En fait, dans ces cas, l'enchaînement des requêtes se fait de telle sorte que, une fois passé un premier stade de placement des données, il n'y a presque plus de mouvement de données au sein de la plate-forme. Les données sont placées de telle sorte que lorsque l'ordonnanceur doit choisir où envoyer une requête, il y a un serveur qui possède déja la donnée et dont le temps d'attente pour l'exécution de cette requête est meilleur que pour les autres serveurs. Ainsi, le placement atteint un état presque stable ou le placement de données est en accord avec leur utilisation. D'ailleurs, nous pouvons vérifier cette diminution avec les valeurs que l'on trouve dans le tableau 3.4. Il contient le nombre de données transférées, le volume correspondant ainsi que le temps nécessaire aux transferts pour la simulation dont le temps d'exécution à été le plus faible. Nous voyons que dans chacun des cas, le nombre de transferts et le volume correspondant est très inférieur à la moyenne constatée. La différence entre le meilleur cas et la moyenne pouvant atteindre un rapport du simple au double.

| | nombre de transfert | volume | temps |
|---|---|---|---|
| LFU | 463 | 98193 Mo | 572 s |
| LRU | 457 | 79571 Mo | 220 s |
| LfuSmall | 495 | 152019 Mo | 1057 s |
| LruSmall | 525 | 166487 Mo | 855 s |
| LfuBig | 383 | 144144 Mo | 700 s |
| LruBig | 401 | 143755 Go | 797 s |

TAB. 3.4 – Information sur les transferts effectués lors des simulations ayant traités 20000 requêtes le plus rapidement pour chacunes des heuristiques testées.

### 3.3.5 Conclusions

Les résultats peu satisfaisants de cette étude préliminaire nous ont conduit à conclure que, dans notre cadre de travail, ne considérer la gestion des données que d'un point de vue cache de données n'était pas suffisant. Dans la majorité des cas, la gestion du cache n'est pas efficace et des données qui vont bientôt être utilisées sont supprimées alors qu'elles auraient du être conservées. Ainsi, les données ne cessent d'être supprimer des caches puis d'être à nouveau téléchargées et stockées ce qui a pour effet de diminuer les performance globales de la plate-forme. Mais, dans de très rares cas, l'ordonnancement des requêtes et la gestion des caches permettent d'atteindre un état où le placement des données se stabilise de lui-même. Dans ces cas, à partir d'un moment dans l'exécution de la plate-forme, l'ordonnancement et le placement deviennent tels qu'il n'est presque plus nécessaire de déplacer des données. Les temps d'attentes dans l'exécution des requêtes liés aux transferts des données devient quasiment nul et l'ensemble de requêtes peut alors être traité beaucoup plus rapidement.

Il vient donc l'idée qu'il pourrait être possible de trouver un placement des données qui permettrait d'avoir la stabilité dès le démarrage de la plate-forme de calcul et de cette façon minimiser les déplacements des données et donc réduire les temps d'accès aux données.

# Chapitre 4

# Algorithmes

## 4.1 Introduction

Dans l'étude préliminaire, nous avons vu sur un cas simple, que gérer le placement des données sans tenir compte de l'ordonnancement ne permettait pas d'exploiter au mieux la capacité d'une plate-forme de façon systématique et prévisible. Nous avons donc décidé d'établir des méthodes qui permettrons de coupler le placement des données à l'ordonnancement en tenant compte des différents paramètres concernant la plate-forme de calcul et son utilisation afin d'atteindre une gestion optimale des ressources de calcul. Dans ce chapitre, nous allons présenter les différents algorithmes et heuristiques que nous avons établis. Dans un premier temps, nous allons poser la modélisation de la plate-forme de calcul ainsi que des divers paramètres décrivant les données et les requêtes qui seront soumises. Ensuite, nous verrons un premier algorithme statique qui détermine un placement des données ainsi que des informations concernant l'ordonnancement des tâches qui seront soumises. Enfin, nous détaillerons des heuristiques permettant une adaptation dynamique du systèmes dans le cas où des changements dans l'utilisation des requêtes et des données se présenteraient.

## 4.2 Modélisation

Nous partons de l'hypothèse que nous avons les objets suivants :
- un ensemble de serveurs de calcul $P_i, i \in [1..m]$,
- un ensemble de données $d_j, j \in [1..n]$,
- un ensemble d'applications $a_k, k \in [1..p]$.

L'ensemble des serveurs de calcul forme la plate-forme qui aura en charge de traiter des requêtes soumises par des utilisateurs. Les serveurs seront donc utilisés pour exécuter des tâches et stocker les données nécessaires à ces tâches. Ainsi, un serveur de calcul $P_i$ doit être caractérisé par deux grandeurs : la première est sa puissance de calcul, noté $w_i$, qui exprime le nombre d'opérations que ce serveur peut effectuer

par seconde. La seconde grandeur est son espace de stockage, noté $m_i$ et exprimé en mégaOctets.

Les données $d_j$ sont des banques de données bioinformatiques publiques stockées dans la plate-forme de calcul et sur lesquelles les utilisateurs vont soumettre des requêtes. Ces données sont accessibles aux utilisateurs uniquement en lecture, ainsi elles ne sont pas modifiées dans le temps. Une donnée $d_j$ est caractérisée par sa taille $taille_j$ exprimée en mégaOctets.

Une application $a_k$ est une application bioinformatique pouvant être exécutée sur n'importe quels serveurs de calcul de la plate-forme. Lors de son exécution celle-ci nécessite une banque de données bioinformatique ainsi que des paramètres d'entrées fournis par l'utilisateur. Typiquement ces paramètres sont une séquence de protéines ou un motif et sont de l'ordre de quelques dizaines d'octets. Au vu des autres grandeurs en jeu – taille des banques, temps d'exécution, nombre de requêtes – nous avons pris le parti de ne pas prendre en compte ces petites données utilisateurs.

Nous appellerons requête (ou tâche) $R_{k,j}$ un couple ($a_k$, $d_j$) où $a_k$ est une application et où $d_j$ est une donnée qui sera utilisée comme entrée à l'application $a_k$. Bien sûr, les banques de données et les applications ayant leurs spécificités ils n'auraient pas de sens de considérer que toutes les banques peuvent servir de données à toutes les applications. Nous définissons alors $v_{k,j} = 1$ si cela a un sens d'appliquer l'algorithme de l'application $a_k$ sur la donnée $d_j$, $v_{k,j} = 0$ sinon. L'analyse des traces d'exécutions du serveur de calcul de l'IBCP nous a permis de déduire que la complexité en temps des applications $a_k$ est affine avec la taille des données. Ainsi, le volume de calcul nécessaire à l'exécution d'une requête de type $R_{k,j}$ est

$$\alpha_k \cdot taille_j + c_k$$

où $\alpha_k$ et $c_k$ sont deux constantes définies pour chacun des algorithmes $a_k$.

Pour chacun des serveurs, nous introduisons également la variable $n_i(k, j)$ qui représente le nombre de requêtes de type $R_{k,j}$ qui seront effectuées sur le serveur de calcul $P_i$. Une requête n'est exécutable sur un serveur que si la banque de données concerné par cette requête est présente sur le serveur.

Enfin, nous définissons $f_{k,j}$ comme la fréquence des requêtes de type $R_{k,j}$ dans le flot de requêtes, elle est exprimé en pourcentage. Comme nous l'avons vu précédemment cette fréquence des requêtes est constante dans le temps.

D'un point de vue du placement des données, il y a deux possibilités pour la plate-forme. Soit l'espace de stockage permet de placer au moins une copie de chacune des données publiques, soit ce n'est pas le cas. Ce second cas peut avoir deux origines : cela peut se produire soit parce que l'espace de stockage total n'est pas suffisant, c'est-à-dire que $\sum_{i=1}^{m} m_i < \sum_{j=1}^{n} taille_j$. Mais la condition sur l'espace de stockage totale disponible dans la plate-forme n'est pas suffisante ; en effet, la répartition de cet espace sur les serveurs peut empêcher la possibilité de stocker certaines grosses données.

Dans la suite nous nous intéresserons uniquement au premier cas, c'est-à-dire que la plate-forme est telle qu'il est possible de stocker au moins une copie de chaque

données dans le système. Notre étude va se porter sur la gestion des données et leur réplication en tenant compte de tous ces paramètres dans le but d'optimiser le temps d'exécution d'un ensemble de requêtes. Ce que nous cherchons à minimiser n'est donc pas le temps de réponse pour une requête particulière, mais le temps global pour un ensemble de requêtes. L'idée est de chercher à utiliser au mieux l'ensemble des ressources disponibles afin d'améliorer le temps de traitement d'un flot de requêtes.

## 4.3 Placement statique des données

### 4.3.1 Introduction

Dans le cadre de la modélisation précédente et à partir des hypothèses que nous avons établi au cours de l'analyse préliminaire concernant l'utilisation des requêtes et des données, nous allons chercher à déterminer un placement statique des données sur les serveurs de la plate-forme. L'objectif de ce placement est d'optimiser le nombre de requêtes traitées sur la plate-forme dans un intervalle de temps donné. Il s'agit là d'un placement statique, c'est-à-dire que les données seront transférées sur les serveurs à l'initialisation du système et ne seront plus déplacées par la suite.

### 4.3.2 Algorithme d'ordonnancement conjoint des calculs et de la réplication

Nous considérerons que nous sommes dans le cas où l'espace de stockage total disponible et la distribution de cet espace sur les serveurs de calcul permet de stocker au moins une copie de chacune des banques de données.

Nous définissons alors $\delta_i^j$, la variable précisant le placement des données : $\delta_i^j = 1$ si la donnée $d_j$ est présente sur le serveur $P_i$, $\delta_i^j = 0$ sinon. Puisque, par hypothèse, nous avons suffisamment de place, nous voulons qu'il y ait au moins un réplicat de chacune des données dans le système.

$$\forall j \sum_{j=1}^{n} \delta_i^j \geqslant 1 \tag{4.1}$$

Mais l'espace sur chaque serveur est limité par sa capacité de stockage. L'ensemble des banques de données stockées sur un serveur ne peut pas dépasser cette capacité.

$$\forall i \sum_{j=1}^{n} \delta_i^j.taille_j \leqslant m_i \tag{4.2}$$

Le nombre de requêtes exécutées sur un serveur $P_i$ est limité par la capacité de calcul de celui-ci. Ainsi, la masse de travail représentée par les requêtes effectuées sur

le serveur $P_i$ ne peut dépasser $w_i$.

$$\forall i \sum_{k=1}^{p} \sum_{j=1}^{n} n_i(k,j)(\alpha_k \cdot taille_j + c_k) \leqslant w_i \tag{4.3}$$

Pour exécuter une requête de type $R_{k,j}$ sur le processeur $P_i$, il faut que la donnée $d_j$ soit présente sur ce processeur. Si ce n'est pas le cas, alors $n_i(k,j)$ doit valoir $0$, dans le cas contraire, $n_i(k,j)$ est borné par le nombre maximal de requêtes $R_{k,j}$ que le serveur est capable d'exécuter.

$$\forall i \; \forall j \; \forall k \; n_i(k,j) \leqslant v_{k,j}.\delta_i^k.\frac{w_i}{\alpha_k.taille_j + c_k} \tag{4.4}$$

Soit $TP$ le rendement de la plate-forme, *i.e.*, le nombre de requêtes qui peuvent être exécutées sur l'ensemble de la plate-forme. La fraction de chaque type de requêtes présentes de l'ensemble des requêtes est définie par $f_{k,j}$. Ainsi, le nombre de requêtes de types $R_{k,j}$ qui sont exécutées doit être limité à cette fraction si l'on ne veut pas défavoriser de type de requêtes et surtout éviter de prendre en compte des requêtes qui n'existeront pas.

$$\forall j \; \forall k \; \sum_{i=1}^{m} n_i(k,j) = f_{k,j}.TP \tag{4.5}$$

En considérant les contraintes exprimées précédemment, notre but est de définir un placement des données qui permettra d'exécuter le plus de requêtes dans un temps donné. C'est-à-dire que nous cherchons à maximiser le rendement $TP$ de la plate-forme. Il en découle le programme linéaire mixte suivant :

**Formulation du programme linéaire**

MAXIMISER $TP$,

SOUS LES CONTRAINTES

$$\begin{cases} (1) & \sum_{j=1}^{n} \delta_i^j \geqslant 1 & 1 \leqslant i \leqslant m \\ (2) & \sum_{j=1}^{n} \delta_i^j.taille_j \leqslant m_i & 1 \leqslant i \leqslant m \\ (3) & n_i(k,j) \leqslant v_{k,j}.\delta_i^j.\frac{w_i}{\alpha_k.taille_j + c_k} & 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n, 1 \leqslant k \leqslant p \\ (4) & \sum_{k=1}^{p} \sum_{j=1}^{n} n_i(k,j)(\alpha_k \cdot taille_j + c_k) \leqslant w_i & 1 \leqslant i \leqslant m \\ (5) & \sum_{i=1}^{m} n_i(k,j) = f_{k,j}.TP & 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n \\ (6) & \delta_i^j \in \{0,1\} & 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n \end{cases}$$

La solution du programme linéaire précédent donne un placement des données sur les serveurs, mais également le nombre de requêtes de chaque type qui doivent être exécutées sur chaque serveur. Nous avons ainsi un placement des données mais aussi des informations sur l'ordonnancement des requêtes. De part l'hypothèse sur l'espace de stockage de données, ce programme linéaire a forcement une solution triviale. En effet, cette hypothèse impose l'existence d'un placement des données permettant de

placer au moins une copie de chaque donnée dans le système. Avec ce placement et en posant $\forall(i,j,k) n_i(k,j) = 0$, nous disposons d'une solution au problème.

Dans la modélisation du système et par conséquent dans le programme linéaire permettant de trouver un placement optimal des données, il peut être surprenant de voir que le réseau d'interconnexion des serveurs, et plus particulièrement la bande passante disponible, ne soit pas pris en compte. En effet, puisque nous parlons de gestion de données, le temps de transfert des données peut sembler un point important à considérer. Mais dans notre étude, pour l'instant, cela n'est pas le cas. Le programme linéaire précédent permet de calculer un placement statique pour les données. Cela signifie qu'une fois que les données seront placées, elles n'auront plus à être déplacées au cours de l'utilisation de la plate-forme. Ainsi, le réseau ne devrait pas être sollicité au cours de l'exécution des requêtes. Il reste alors le problème de l'initialisation du système et du placement initiale. Puisque nous sommes dans un modèle en régime permanent, nous avons choisi de ne pas considérer les coûts de transferts initiaux.

### 4.3.3 NP-complétude du problème

**Définition 4.1** (SCHEDULE-REPLICATION-DEC). *Étant donnés $m$ processeurs $P_i, i \in [1..m]$ ayant une puissance de calcul $w_i$ et un espace de stockage $m_i$, $n$ données $d_j$ de taille $size_j$, $p$ applications $a_k$ qui peuvent être appliquées sur les données $d_j$, $v_{k,j}$ définissant si l'application $a_k$ peut être appliquée sur la donnée $d_j$, $\alpha_k$ et $c_k$ tel que $\alpha_k \cdot size_j + c_k$ est la somme de calcul nécessaire pour exécuter l'application $a_k$ sur la donnée $d_j$, $f_(k,j)$ la proportion des requêtes de $a_k$ sur $d_j$ et une borne $K$, est-il possible de trouver un placement $\delta_i^j$ et un ordonnancement $n_i(k,j)$ tel que le débit $TP$ soit supérieur ou égal à $K$ ?*

**Théorème 4.1.** *Le problème SCHEDULE-REPLICATION-DEC est NP-complet.*

*Démonstration.* Tout d'abord, de façon triviale, le problème SCHEDULE-REPLICATION-DEC appartient à NP. En effet, nous voyons facilement que vérifier qu'un placement et un ordonnancement sont solutions du problème peut se faire en temps polynomial par rapport à $n \cdot m \cdot p$. Pour prouver qu'il s'agit d'un problème NP-complet, nous allons procéder par réduction au problème 2-PARTITION connu pour être NP-complet [20].

Une instance $\mathcal{I}_1$ arbitraire du problème 2-PARTITION serait la suivante : étant donnés $N$ entiers positifs $a_i, 1 \leqslant i \leqslant N$ et un entier positif $S$ tel que $2S = \sum_i a_i$, est-il possible de trouver un ensemble $I \subset \{1..N\}$ tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i = S$ ?

Nous construisons alors l'instance $\mathcal{I}_2$ de SCHEDULE-REPLICATION qui suit : soit $N$ données de taille $size_j = a_j$, $P_1$ et $P_2$ deux processeurs tels que $w_1 = w_2 = S + N$ et $m_1 = m_2 = S$, une application $a_1$ avec pour paramètres $\alpha_1 = 1$ et $c_1 = 1$, $f_(1,j) = \frac{1}{N}$, $v_{1,j} = 1$ et $K = N$. La construction d'une telle instance $\mathcal{I}_2$ en partant de $\mathcal{I}_1$ est polynomiale. Nous allons montrer que $\mathcal{I}_2$ admet une solution si et seulement si $\mathcal{I}_1$ en admet une.

– Supposons tout d'abord que $\mathcal{I}_1$ admet une solution et posons $I$ un sous-ensemble de $\{1..N\}$. Pour tout $j \in I$, nous définissons $\delta_1^j = 1$, $\delta_2^j = 0$, $n_1(1,j) = 1$ et $n_2(1,j) = 0$. Pour tout $j \notin I$, nous définissons $\delta_1^j = 0$, $\delta_2^j = 1$, $n_1(1,j) = 0$ et $n_2(1,j) = 1$. Avec un tel placement, toutes les données sont placées sur au moins un processeur et la taille des données stockées par un processeur vérifie :

$$\sum_{j=1}^{N} \delta_1^j \cdot size_j = \sum_{j \in I} size_j = \sum_{j \in I} a_j = S$$

Le volume de calcul sur chaque processeur est

$$\sum_{j=1}^{N} n_1(1,j)(\alpha_k \cdot size_j + c_k) = \sum_{j \in I}(a_j + 1) = S + \sum_{j \in I} 1 \leqslant S + N$$

Les mêmes résultats sont obtenus pour le processeur $P_2$. Le débit de la plateforme est égal à

$$TP = \sum_{i=1}^{2}\sum_{j=1}^{N} n_i(1,j) = \sum_{j=1}^{N} n_1(1,j) + \sum_{j=1}^{N} n_2(1,j) = \sum_{j \in I} 1 + \sum_{j \notin I} 1 \geqslant N$$

Par conséquent $TP \geqslant N$ et $\mathcal{I}_2$ a une solution.

– Supposons maintenant que $\mathcal{I}_2$ a une solution, et posons $\delta_i^j$ et $n_i(k,j)$ les placements et ordonnancement de cette solution. Sur chaque processeur, l'espace occupé par les données stockées ne peut dépasser $m_i$, ainsi

$$\sum_{j=1}^{N} \delta_1^j \cdot size_j \leqslant m_1.$$

$$\sum_{j=1}^{N} \delta_2^j \cdot size_j \leqslant m_2.$$

Mais $size_j = a_j$, $m_1 = m_2 = S$ et $\sum_j a_j = 2S$ alors, en ajoutant les deux inéquations :

$$\sum_{j=1}^{N}(\delta_1^j + \delta_2^j) \cdot a_j \leqslant 2S$$

$$\sum_{j=1}^{N}(\delta_1^j + \delta_2^j) \cdot a_j \leqslant \sum_{j=1}^{N} a_j$$

Mais $\delta_i^j \in \{0,1\}$ et chaque donnée est placée au moins une fois dans la plateforme, nous avons donc :

$$\forall j \in \{1..N\} \quad \delta_1^j + \delta_2^j \geqslant 1$$

ainsi $\forall j \in \{1..N\}$ soit $\delta_1^j = 1$ et $\delta_2^j = 0$ soit $\delta_1^j = 0$ et $\delta_2^j = 1$. Posons alors $I = \{j, \delta_1^j = 1\}$, et nous obtenons

$$\sum_{j \in I} a_j = \sum_{j=1}^{N} \delta_1^j \cdot a_j = S$$

et $\mathcal{I}_1$ a une solution.

■

### 4.3.4 Approximation entière

Comme $\delta_i^j$ vaut $0$ ou $1$, le programme linéaire est mixte, entier et rationnel. Dans la section précédente, nous avons prouvé que ce problème est NP-complet par une réduction simple au problème 2-PARTITION. Cela signifie principalement, que la complexité du programme nécessaire pour déterminer une solution optimale est telle que cela ne peut pas être effectué dans un temps raisonnable. Pour cette raison, nous avons défini une heuristique utilisant les mêmes paramètres que ceux nécessaire à l'écriture du programme linéaire dont l'objectif est de déterminer une solution proche de la solution optimale dans un temps borné. Nous avons donc opté pour une résolution du programme linéaire relâché en rationnel, puis, par approximations successives, nous construisons une solution entière.

L'algorithme 4.1 décrit notre méthode pour atteindre une solution entière à partir de la solution rationnelle. L'idée générale est de commencer par placer au moins une fois chacune des données, et ensuite de placer en priorité les données les plus consommatrices en terme de volume de calculs. Les hypothèses de départ nous permettent de calculer le volume de calcul nécessaire à l'exécution d'une requête. En multipliant ce volume par la fréquence d'apparition de ce type de requêtes, nous obtenons la fraction de volume de calcul qu'implique ce type de requêtes dans le flot total. Ainsi, pour une donnée, si l'on additionne ces volumes pour tous les types de requêtes qui la concernent, nous obtenons ce que nous avons appelé la consommation en terme de volume de calculs d'une donnée $vol[d_j]$

$$vol[d_j] = \sum_{k=1}^{p} f(k,j) \cdot (\alpha_k \cdot m_j + c_k)$$

Le départ de l'algorithme se fait à partir de la solution rationnelle du programme linéaire. Nous construisons alors trois ensembles : $S_0$ qui contient les couples $(i,j)$ pour lesquels $\delta_i^j = 0$, $S_1$ qui contient les $(i,j)$ où $\delta_i^j = 1$ et $S$ pour tous les $(i,j)$ qui ne sont ni dans $S_0$ ni dans $S_1$. Ensuite, nous déterminons l'ensemble $notPlaced$ des banques de données dont aucun réplicat n'a encore été placé. Si cet ensemble n'est pas vide, on choisit un couple $(i_1, j_1)$ avec $j_1 \in notPlaced$ et tel que la consommation en terme de calcul de cette donnée multiplié par la valeur courante de $\delta_{i_1}^{j_1}$ soit la plus grande. Si $notPlaced$ est vide, on choisit alors un couple $(i_1, j_1)$ dans $S$ avec la même contrainte.

Dans les deux cas, le couple $(i_1, j_1)$ est ajouté à $S_1$. Alors pour tous les couples $(i, j)$ dans $S_1$, on rajoute la contrainte $\delta_i^j = 1$ au programme linéaire original. On pratique de même en ajoutant la contrainte $\delta_i^j = 0$ pour les couples dans $S_0$. Enfin, on résout ce nouveau programme linéaire. Si ce nouveau programme linéaire n'a pas de solution, altors on retire le couple choisi $(i_1, j_1)$ de $S_1$ et on le place dans $S_0$. On reconstruit alors le programme linéaire et on le résout.

À partir de la nouvelle solution, on réitère toutes les étapes précédentes tant que $S$ est non vide.

**Algorithme 4.1** Algorithme d'approximation d'une solution entière

1: Résoudre le programme linéaire rationel $lp$
2: Soit $S_0 = \{(i, j) | \delta_i^j = 0\}$
3: Soit $S_1 = \{(i, j) | \delta_i^j = 1\}$
4: Soit $S = \{(i, j) | (i, j) \notin S_0 and (i, j) \notin S_1\}$
5: Soit $notPlaced = \{j \in [1..n] | \forall i \in [1..m] \delta_i^j \neq 1\}$
6: **Si** $notPlaced \neq \emptyset$ **Alors**
7: $(i_1, j_1) | (\sum_{k=1}^{p} f(k, j) \cdot (m_{j_1} \cdot \alpha_k + c_k) \cdot \delta_{i_1}^{j_1} = max_{\{j \in notPlaced, (i,j) \in S\}} \sum_{k=1}^{p} f(j, k) \cdot (m_j \cdot \alpha_k + c_k) \cdot \delta_i^j$
8: Ajouter $(i_1, j_1)$ à $S_1$
9: **Sinon**
10: $(i_1, j_1) | (\sum_{k=1}^{p} f(k, j) \cdot (m_{j_1} \cdot \alpha_k + c_k) \cdot \delta_{i_1}^{j_1} = max_{\{(i,j) \in S\}} \sum_{k=1}^{p} f(k, j) \cdot (m_j \cdot \alpha_k + c_k) \cdot \delta_i^j$
11: Ajouter $(i_1, j_1)$ à $S_1$
12: **Pour tout** $\delta_i^j \in S_1$ **:**
13: Ajouter la contrainte $\delta_i^j = 1$ à $lp$
14: **Pour tout** $\delta_i^j \in S_0$ **:**
15: Ajouter la contrainte $\delta_i^j = 0$ à $lp$
16: Résoudre le nouveau $lp$
17: **Si** $lp$ n'a pas de solution **Alors**
18: Retirer $(i_1, j_1)$ de $S_1$
19: Ajouter $(i_1, j_1)$ à $S_0$
20: Refaire les étapes 13 à 18 et résoudre
21: Recommencer à l'étape 2 tant que $S \neq \emptyset$

### 4.3.5 Une approche gloutonne au problème du placement

À partir des mêmes données de la modélisation concernant la plate-forme et des requêtes, nous avons défini un algorithme glouton 4.2 pour résoudre le problème du placement. Le principe de cet algorithme est d'essayer de placer les données les plus consommatrices en terme de calculs sur les serveurs qui ont les capacités de calcul les plus élevées.

L'algorithme commence par calculer la consommation en terme de volume de calcul pour chaque donnée. On trie alors les données par valeur décroissante de ce volume. On trie également les serveurs par leur capacité de calcul décroissante. Nous allons alors essayer de placer la donnée qui nécessite le plus gros volume de calcul sur le serveur disposant de la plus grosse puissance de calcul. Si ce serveur ne dispose pas d'assez de place, on essaie de placer la donnée sur le serveur suivant dans la liste triée, et ainsi de suite jusqu'à trouver un serveur ayant suffisamment d'espace de stockage libre. On continue en essayant de placer la deuxième donnée, en terme de volume de calcul nécessaire, sur le premier serveur, en terme de puissance de calcul. De même, s'il n'y a pas la place, on continue en parcourant la liste des serveurs jusqu'à en trouver un avec suffisamment de place. On procède de même pour toutes les données. Une fois toutes les données placées au moins une fois, on recommence au début de la liste des données et ainsi de suite jusqu'à ce qu'il ne soit plus possible de placer aucune donnée.

**Algorithme 4.2** Algorithme glouton pour le placement des données

**Pour tout** $d_j$ **:**
  $s_j = \sum_{k=1}^{p} f(j,k) \cdot (\alpha_k \cdot m_j + c_k)$
Trier les $d_j$ par valeur décroissante de $s_j$ dans $sortData$
Trier les $P_i$ par valeur décroissante de $w_i$ dans $sortServer$
**Tant que** il y a suffisamment de place pour une donnée **:**
  **Pour** $j = 0 ; j < n ; j++$ **:**
    $i = 0, place = false$
    **Tant que** $!place$ and $i < n$ **:**
      **Si** il y a assez de place sur $sortServer[i]$ pour $sortData[j]$ **Alors**
        **Si** $sortData[j]$ n'est pas déjà sur $sortServer[i]$ **Alors**
          Placer cette donnée sur ce serveur
          $place = true$
        $i++$

### 4.3.6 Remarques sur la solution

Afin d'effectuer une première vérification de la cohérence des solutions obtenues grâce au programme linéaire, nous avons exécuté celui-ci en utilisant comme entrées des données extraites des traces de NPS@ pour toutes les informations concernant les banques de données, les applications et leur utilisation. Pour la description de la plate-forme, nous avons défini une plate-forme avec une dizaine de nœuds dont nous avons fait varier les paramètres de puissance et de stockage pour voir les répercussions. Il ne s'agissait pas là de faire une étude des performances obtenues par l'algorithme mais bien de vérifier qu'il donnait des résultats cohérents et qu'il n'y avait pas eu d'oublis de contraintes lors de sa conception.

La première constatation que nous pouvons faire est que le placement obtenu ne

remplit pas forcement l'intégralité de l'espace de stockage disponible, même lorsqu'il serait encore possible de créer des réplicats de certaines données sur des serveurs sans créer de doublons sur ceux-ci. Cela s'explique assez facilement par le fait que le placement des données est, par les contraintes définies, intimement lié aux capacités de calcul des serveurs. Ainsi, créer des réplicats pour les données peu utilisées est inutile puisqu'ils ne serviront à rien. La deuxième constatation est que dans certains cas extrêmes où l'espace de stockage est vraiment restreint, alors la capacité de traitement de tous les serveurs n'est pas forcément atteinte. La raison est la même que précédemment : en effet, calcul et stockage étant liés, il est possible que l'espace de stockage ne permette pas de stocker suffisamment de données pour engendrer un volume de calcul capable de saturer le processeur. Enfin, la dernière constatation est que dans la grande majorité des cas que nous avons testés, la valeur de la fonction objectif avec la solution rationnelle est similaire, ou très proche de la valeur de cette même fonction avec la solution rationnelle. Cela tend à prouver que notre algorithme d'approximation de la solution entière prend en compte les bons paramètres afin d'approcher une solution optimale.

## 4.4 Redistribution dynamique

### 4.4.1 Introduction

Dans la section précédente, nous avons donné une méthode permettant de trouver un placement proche de l'optimal en fonction de conditions initiales fixes. Cet algorithme part du principe que l'utilisation des données et des algorithmes est connue à l'avance. L'étude des traces d'exécutions d'un serveur existant nous a permis de montrer que cette hypothèse n'était pas irréaliste et était un bon point de départ. Mais les données que nous considérons sont des banques de données biologiques principalement utilisées par des chercheurs en bioinformatique. Il n'est donc pas impossible que, dans le cadre d'une expérience bien particulière, une banque de données usuellement peu utilisée soit subitement la cible d'un grand nombre de requêtes. Dans l'algorithme statique de la version précédente, tous les calculs de placement des données et des informations d'ordonnancement sont faits avant la soumission réelle des requêtes dans la plate-forme. De cette façon, cet algorithme n'est pas capable de s'adapter à un changement des schémas d'utilisation, puisqu'une fois que le placement est fait, il n'est jamais remis en cause.

Pourtant, l'idée de départ consistant à dire que globalement l'utilisation est constante reste cohérente. Nous avons donc étudié comment rendre cette méthode dynamique afin d'être en mesure de s'adapter. Pour cela, il faut d'abord être en mesure de détecter les changements d'utilisation de la plate-forme, et surtout les éléments qui vont faire diminuer les performances globales de la plate-forme.

### 4.4.2 Détecter et décider

Nous sommes toujours partis de l'hypothèse que nous avions suffisamment de requêtes pour saturer l'ensemble de la plate-forme cible en calcul et donc que tous les processeurs disponibles sont toujours employés. Dans ce contexte, une utilisation optimale de l'ensemble de la grille de calcul implique que l'ensemble des nœuds a une charge de travail équivalente en terme de temps de calcul. C'est-à-dire que pour deux nœuds distincts de la grille, le temps de calcul que représente l'ensemble des requêtes placées dans leur file d'attente devrait être relativement équivalent. En effet, prenons un cas simple avec un ordonnancement sur deux processeurs, et considérons que l'un des deux processeurs a un temps de calcul très largement supérieur à l'autre. Alors, le second processeur aura fini de traiter son lot de travail largement avant l'autre et n'aura plus rien à traiter alors que le premier processeur aura encore du travail à effectuer. Ainsi, le deuxième processeur aurait pu traiter une partie du travail de l'autre et l'ensemble des requêtes aurait été traité plus rapidement. Les figures 4.1 et 4.2 sont une illustration de ce raisonnement. Dans la première figure, les deux serveurs dont les tailles des files de requêtes sont déséquilibrés. Dans la seconde figure, le même ensemble de tâches est reparti équitablement entre les deux files et le temps de traitement des requêtes est amélioré. Il est facile d'étendre cette constatation avec un nombre plus important de processeurs. Bien sûr, dans un contexte très spécifique, par exemple avec un espace de stockage très fortement restreint et donc de fortes contraintes sur le placement des données, il est possible que de tels déséquilibres soient tout à fait normaux. Il peut se produire aussi des décalages si les temps de calcul entre deux types de requêtes sont vraiment disproportionnés et qu'une seule requête extrêmement longue à exécuter soit seule cause du déséquilibre. Mais il s'agit là de cas pathologiques, qui bien que pouvant mériter une attention particulière, sortent légèrement du cadre de notre étude. En effet, une des caractéristiques des applications bioinformatiques qui servent de support à nos travaux est d'être de courte durée (quelques minutes), le problème d'optimisation des performances ne se situant alors pas sur un petit nombre de requêtes très coûteuses en temps mais plutôt sur de très grands ensembles de requêtes de courte durée. Globalement, lorsque les ressources sont suffisantes pour effectuer un placement efficace, alors la charge de travail de chaque serveur devrait être relativement équilibrée.

Le point que nous allons donc chercher à surveiller est la taille, en terme de temps de calcul, des files d'attente de requêtes de chaque nœud de la grille. Si une de ces files venait à être nettement plus importante que les autres, cela signifierait qu'il y a un problème dans l'ordonnancement, et donc qu'il est nécessaire d'effectuer une correction sur le système. De la même façon, l'existence d'un, ou plusieurs nœuds, sous-chargés par rapport aux autres peut signifier qu'il soit possible de rééquilibrer la charge.

Pour effectuer cette opération de surveillance de la plate forme, nous avons rajouté un module dans l'architecture fonctionnelle de l'ensemble de notre système. La figure 4.3 montre l'architecture fonctionnelle de l'ensemble du système. Ce nou-

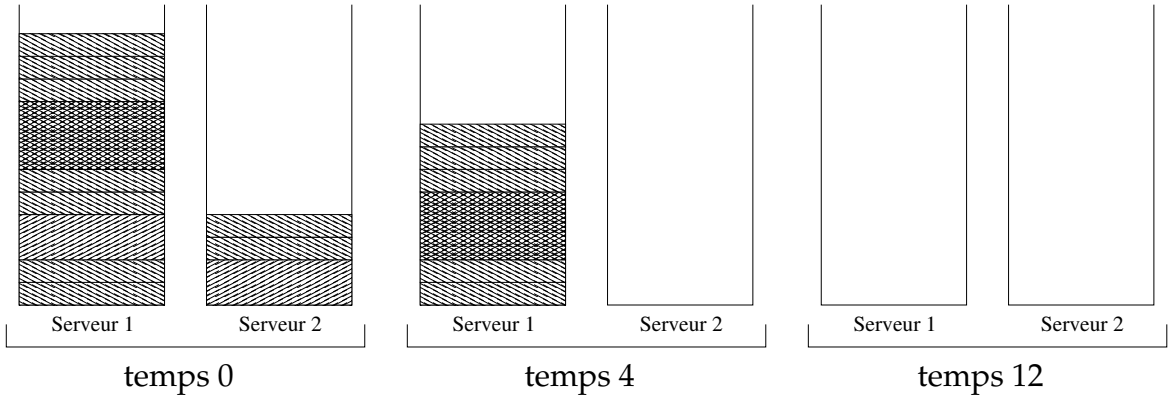


FIG. 4.1 – Exécution d'un ensemble de tâches sur deux serveurs dont les files de requêtes n'ont pas la même taille. Le serveur 2 finit son travail avant le serveur 1.

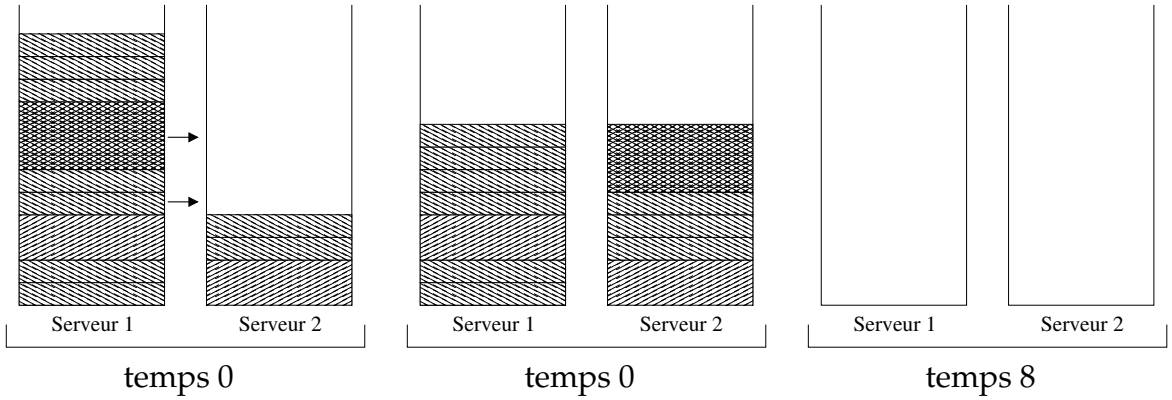


FIG. 4.2 – Exécution du même ensemble de tâches que la figure précédente, mais on équilibre les deux files de requêtes pour que les deux serveurs aient la même somme de travail. Les deux serveurs finissent alors simultanément leurs tâches et l'ensemble des requêtes à été traité plus rapidement que dans l'exemple de la figure 4.1.

veau module, appelé DynSraManager interrogera à intervalle régulier l'ensemble des nœuds disponible afin de collecter des informations sur leur charge de travail. A partir de cela, il est possible de déterminer la charge moyenne de l'ensemble des nœuds. Un bon ordonnancement devrait faire en sorte que tous les nœuds aient globalement la même charge.

L'opération de rééquilibrage de charge entre les nœuds se déroule donc en plusieurs phases. Tout d'abord, il faut déterminer les nœuds qui sont en surcharge de travail par rapport aux autres. Ensuite, il faut trouver quelles sont les causes de cette surcharge et enfin trouver des solutions à apporter pour corriger ce point, et améliorer les performances globales en fonction des conditions actuelles. Pour tous ces points nous avons étudié différentes heuristiques.

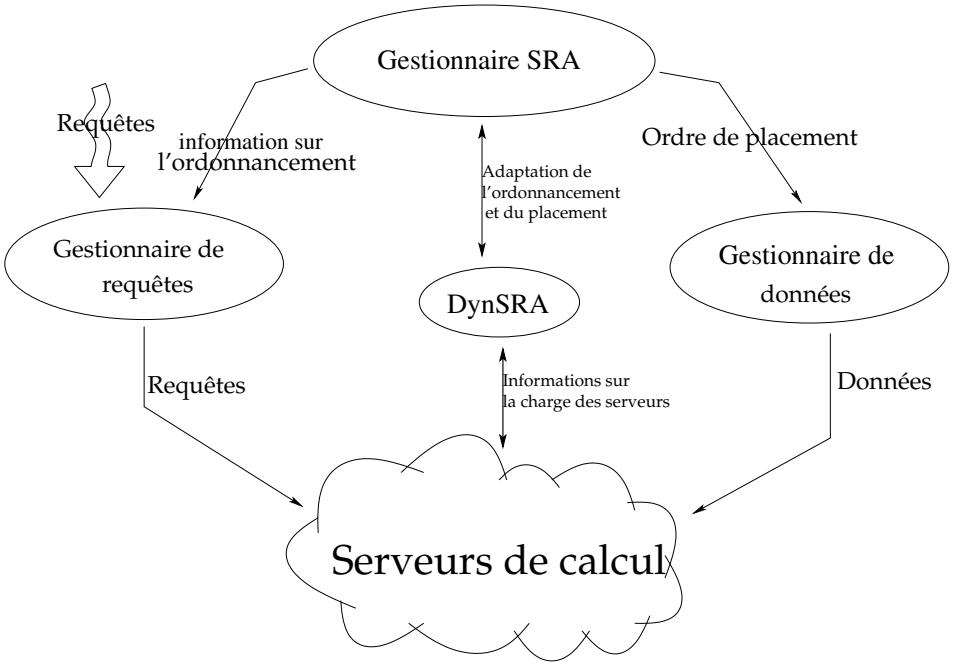


FIG. 4.3 – Architecture fonctionnelle liée à DynSRA.

### 4.4.3 Les nœuds en surcharge

Dire qu'un nœud est en surcharge par rapport à un ensemble de nœuds peut se décider de différentes manières car la notion de charge est assez complexe. Puisque nous nous plaçons dans un contexte de nœuds hétérogènes avec des tâches et des données de tailles diverses, se contenter de regarder le nombre de requêtes dans les files d'attente n'est pas satisfaisant. En effet, un nœud ayant d'importantes ressources de calcul traitera plus rapidement une même suite de requête qu'un nœud de moindre puissance. Puisque nous connaissons les caractéristiques, en terme de coût de calcul, de chacun des algorithmes, ainsi que la capacité de traitement des nœuds, nous pouvons donc facilement déterminer une bonne approximation du volume de calcul induit par l'exécution d'un ensemble de requêtes, et plus particulièrement par celles présentes dans la file d'attente d'un nœud. Ainsi, nous pouvons considérer comme en surcharge un nœud dont le volume de calcul de l'ensemble des requêtes dans la file est le plus important. Si l'on pose $N(i,j,k)$ le nombre de requêtes de type $R(j,k)$ dans la file d'attente du nœud $P_i$, nous pouvons définir le serveur en surcharge $P_{i_{surcharge}}$ tel que :

$$W_{i_{surcharge}} = max_i \{W_i\}$$

où

$$W_i = \sum_j \sum_k (N(i,j,k) \cdot (\alpha_k \cdot size_j + c_k))$$

est le volume de calcul induit par le traitement des requêtes dans la file d'attente du serveur $P_i$. Nous avons appelé **MaxCompServer** cette méthode pour choisir le serveur le plus chargé. **MaxCompServ** ne se base que sur le volume de calcul et ne tient donc pas compte de la puissance des nœuds. Or, dans une plate-forme hétérogène, les différences de puissances entre les nœuds peut avoir un impact significatif sur le temps qui sera nécessaire au traitement des calculs. Ce n'est pas forcement le nœud qui a le plus de calcul à effectuer qui mettra le plus temps pour accomplir son travail.

Ainsi, il faudrait introduire la puissance de calcul dans la mesure de la surcharge et considérer plutôt le temps nécessaire pour effectuer l'ensemble des requêtes dans la file d'attente. De cette façon, les nœuds considérés en surcharge sont ceux qui vont avoir trop de travail par rapport à leur capacité de traitement. Ce ce que nous prenons en compte dans l'heuristique **MaxTimeServ** dans laquelle le processeur $P_{i_{surcharge}}$ en surcharge est tel que :

$$\frac{1}{w_{i_{surcharge}}} \cdot W_{i_{surcharge}} = max_i \left\{ \frac{1}{w_i} \cdot W_i \right\}$$

Cette méthode semble offrir un moyen de comparaison de la charge des nœuds qui semble plus équitable au vu des différences de puissance de calcul. Mais, nous perdons alors le lien avec la notion de volume de calcul qui est pourtant la base de la surcharge.

Avec l'heuristique de choix **MaxOverMedianServ**, nous avons chercher à combiner les deux méthodes précédentes. Le lien entre temps de calcul et volume de calcul est très simple puisque, pour un nœud $P_i$, ces deux valeurs sont liées par $w_i$. En effet, si $T_i$ est le temps d'exécution nécessaire pour traiter les données dans la file de $P_i$ alors :

$$T_i = \frac{W_i}{w_i}$$

Ainsi, avec **MaxOverMedianServ**, nous choisissons de définir le serveur surchargé comme étant celui dont le temps de calcul des requêtes dans sa file est séparé de la moyenne par le plus grand volume de calcul. Autrement dit, pour chaque serveur, nous calculons le temps de calcul nécessaire au traitement des requêtes dans sa file d'attente. Nous pouvons alors déterminer le temps moyen pour l'ensemble des serveurs. Enfin, pour chaque serveur, il est possible de calculer le volume de calcul dont le temps d'exécution représente la différence entre la moyenne et son temps d'exécution des requêtes en queue. Le nœud le plus chargé sera alors celui dont le volume de calcul ainsi défini sera le plus important. C'est-à-dire que le nœud $P_{i_{surcharge}}$ est tel que

$$(T_{i_{surcharge}} - T_{moyen}) \cdot w_{i_{surcharge}} = max_i \left\{ (T_i - T_{moyen}) \cdot w_i \right\}$$

où $T_{moyen} = \frac{1}{m} \sum_{i=1}^{m} T_i$ est le temps de calcul moyen sur l'ensemble des nœuds. En agissant de la sorte, nous ne perdons pas de vue que c'est un volume de calcul qui est la cause du problème de surcharge et surtout que c'est ce volume dont il faudra réussir à répartir sur d'autres nœuds.

**Remarque** Cependant quelque soit l'heuristique utilisée, le nœud choisi n'est pas forcement un nœud réellement en surcharge. Si tous les nœuds sont pratiquement au même niveau de charge, une telle méthode de détermination donnera tout de même un nœud surchargé alors que ça n'est pas forcément le cas. Nous avons donc décidé de ne considérer que les nœuds donc la charge est supérieure à un seuil. Ce seuil est défini en fonction de la moyenne des charges de l'ensemble des nœuds.

### 4.4.4 Les causes de la surcharge

Une fois qu'un nœud a été déterminé comme étant en surcharge, il faut trouver la cause de la surcharge et voir comment il est possible de la réduire. Puisque, par hypothèse, tous les nœuds sont capables d'exécuter tous les algorithmes, les points sur lesquels nous pouvons jouer pour délester un nœud d'une partie de sa charge sont l'ordonnancement et le placement des données. En effet, en réplicant une donnée sur un ou plusieurs serveur, il va être possible de répartir les requêtes portant sur cette donnée sur un plus grand nombre de serveurs. De même, si une donnée est déja présente sur un serveur en sous-charge, modifier l'ordonnancement des requêtes afin que se serveur se voit attribué une plus grande part des requêtes concernant cette donnée devrait permettre de délester d'une partie de son travail un serveur surchargé. Bien évidemment, ce rééquilibrage ne peut fonctionner que si les requêtes à exécuter sont de courtes durées, c'est-à-dire que la surcharge n'est pas provoqué par une unique requête mais par un trop grand nombre de petites requêtes. Typiquement, dans la catégorie d'applications bioinformatiques que nous avons étudiée c'est ce comportement que l'on retrouve, à savoir un très grand nombre de requêtes de courte durée (quelques minutes).

Le problème consiste donc à trouver la ou les données qui provoquent la surcharge et voir s'il est possible de décharger une partie des requêtes qui les concernent sur un autre nœud. Pour cela, la première étape est de trouver les données responsable de la surcharge. Comme pour la détermination des nœuds en surcharge plusieurs méthodes sont possibles.

Là encore, au vue des hypothèses et des informations déjà connues sur l'ensemble des algorithmes, données et serveurs, il est aisé, pour chaque nœud, de calculer à un instant donné le volume de calcul engendré par une donnée sur un nœud. Pour le nœud $i$, le travail $w(i,j)$ dont la donnée $d_j$ est responsable est :

$$w(i,j) = \sum_k \left(N(i,j,k) \cdot (\alpha_k \cdot size_j + c_k)\right)$$

Ainsi, sur le serveur surchargé nous pouvons considérer que la donnée responsable de la surcharge est celle qui à un $w(i,j)$ maximum. Nous appelerons **MaxCompData** l'heuristique qui consiste à choisir la donnée à répliquer selon ce critère. Retirer au nœud une partie des requêtes sur cette donnée aura pour conséquence de réduire la charge de ce nœud.

Cependant, ce n'est pas parce qu'une donnée représente la plus grande partie de la charge d'un nœud que cette donnée est responsable de la surcharge. En effet, une surcharge peut être dûe à une utilisation d'une donnée qui n'est pas celle qui avait été prévue initialement et dont les fréquences d'utilisation diffèrent de celles qui ont servi à l'établissement du placement originel. Dans ce sens, au lieu de ne considérer que la charge des nœuds, une idée serait de confronter l'utilisation réelle des données avec les prévisions statistiques de départ. Ainsi, les données dont l'utilisation diffère le plus des prévisions initiales seront celles sur lesquelles une attention toute particulière devra être portée afin de prendre en compte les modifications d'utilisations. Cela implique que le module responsable de la surveillance tienne à jour un historique des requêtes effectuées sur chaque nœud afin de pouvoir faire la comparaison. Si $f(j,k)$ est définie comme la fréquence d'apparition théorique de la tâche $R(j,k)$ dans le flot de requêtes, nous définissons $f_{reel}(j,k)$ la fréquence réelle dans le flot de requêtes reçu par l'ordonnanceur. Nous définissons alors l'heuristique **MaxDiffData**, comme étant celle qui choisit la donnée dont l'utilisation réelle est la plus différente de ce qui avait été prévue. La donnée $d_{j_{surcharge}}$ est alors la donnée telle que

$$\sum_k \left(f_{reel}(j_{surcharge},k) - f_(j_{surcharge},k)\right) = max_j \left(\sum_k \left(f_{reel}(j,k) - f_(j,k)\right)\right)$$

### 4.4.5 Redistribuer les volumes de calcul

De la même façon que l'on peut estimer un nœud en surcharge, on peut définir un nœud comme étant sous-chargé, c'est-à-dire ayant moins de travail à effectuer en comparaison des autres nœuds. En prenant l'opposé des mesures de surcharge décrit dans la section 4.4.3, il est possible de définir des heuristiques pour déterminer la sous-charge. Si l'on considère le volume de calcul, cela signifie que le nœud le plus chargé est tel que :

Avec les mêmes notations que précédemment, en considérant les volumes de calcul, le nœud $P_{i_{sous-charge}}$ est définit tel que :

$$W_{i_{sous-charge}} = min_i \{W_i\}$$

Nous nommerons **MinCompServ** cette heuristique.

Si l'on considère cette fois les temps de calcul, l'heuristique **MinTimeServ** définit $P_{i_{soucharge}}$ par :

$$\frac{1}{w_{i_{souscharge}}} \cdot W_{i_{souscharge}} = min_i \left\{\frac{1}{w_i} \cdot W_i\right\}$$

Enfin, dans **MaxUnderMedianServ**, nous considérons le volume et le temps de calcul des requêtes dans la file d'un serveur en évaluant les distances en terme de volume de calcul au temps de calcul moyen. Dans ce cas, $P_{i_{souscharge}}$ est alors tel que

$$(T_{i_{souscharge}} - T_{moyen}) \cdot w_{i_{souscharge}} = min_i \{(T_i - T_{moyen}) \cdot w_i \}$$

**Remarque** Comme pour le choix des serveurs en surcharge, un seuil doit être définit pour pouvoir réellement déclarer un serveur comme étant sous-chargé.

### 4.4.6 Ajout et suppression de données

Considérons que nous avons déterminé un nœud $P_{i_{surcharge}}$ comme étant en surcharge, la donnée $d_{j_{surcharge}}$ stockée sur $P_{i_{surcharg}}$ comme étant responsable de la surcharge et un nœud $P_{i_{soucharge}}$ en sous-charge. L'idée est alors de confier à $P_{i_{soucharge}}$ une partie des calculs de $P_{i_{surcharge}}$ liés à $d_{j_{soucharge}}$. Si $P_{i_{soucharge}}$ possède déjà la donnée concernée, alors il suffira de mettre à jour l'ordonnancement comme cela sera expliqué dans la section suivante. Dans le cas où le serveur en sous-charge n'a pas la donnée et qu'il a encore suffisamment de place pour la stocker, il suffit de créer le réplicat et de mettre l'ordonnanceur à jour. Mais si le serveur $P_{i_i soucharge}$ n'a ni la donnée, ni suffisamment de place pour la réplication, il peut être envisagé de supprimer une ou plusieurs de ses données pour placer un nouveau réplicat de $d_{j_{surcharge}}$

Pour cela, il faut choisir une donnée sur le serveur en sous-charge pour laquelle les répercussions sur le reste des autres serveurs de calcul seront moindres. Une première méthode, que nous nommerons **MinCompData**, consiste à considérer le volume de calcul engendré par chaque donnée présente sur le serveur considéré et choisir celle qui minimise cette valeur. Ainsi, $d_{j_{sous-charge}}$ serait la donnée telle que :

$$w_{i_{sous-charge},j_{sous-charge}} = min_{i,j} \{w_{i,j}\}$$

Une deuxième méthode, appelée **LessUsedData**, consiste à observer l'utilisation de chacune des données dans le temps plutôt que son utilisation à l'instant donnée. Une donnée peu utilisée serait alors définie comme celle dont la fréquence d'utilisation locale aurait la valeur la plus basse dans l'historique des requêtes soumises sur ce nœud.

Toutefois, avant de déclencher le processus de suppression d'une donnée, il est nécessaire de vérifier que cette donnée est toujours présente sur au moins un autre serveur. Dans le cas contraire, nous risquerions de nous retrouver avec des requêtes impossibles à satisfaire car il existerait dans le système des requêtes sur une donnée n'étant plus présente sur aucun des serveurs de calculs. Si la donnée n'a pas de réplicat, il est alors impossible de la supprimer. Il faudra donc voir s'il est possible de supprimer une autre donnée sur ce serveur, et si aucune donnée ne peut être supprimée, il faudra envisager de se servir d'un autre serveur en sous-charge pour le processus.

### 4.4.7 Réordonnancer

Une fois que nous avons défini un nœud $P_{i_{surscharge}}$ en surcharge, une donnée $d_j$ dont on va chercher à répartir une partie du travail qu'elle implique sur un serveur en sous-charge $P_{i_{sous-charge}}$, il faut mettre à jour les informations de l'ordonnanceur afin de

tenir compte au mieux de cette évolution de la plate-forme. Là encore, cette opération peut être effectuée de plusieurs façons.

Une première méthode, **EqShareSched**, assez basique consiste à partager à part égale entre le serveur sous-chargé et le serveur surchargé les requêtes impliquant la donnée concernée. C'est-à-dire que si $n(i,j,k)$ est la valeur d'ordonnancement pour le serveur $P_i$ pour le type de requête $R(j,k)$, on va adapter les valeurs pour les nœuds $P_{i_{surcharge}}$ et $P_{i_{sous-charge}}$, tels que :

$$\begin{array}{ll} \forall k, v(j,k) \neq 0 : & n(i_{sous-charge},j,k) = n(i_{sous-charge},j,k) + \frac{1}{2} \cdot n(i_{surcharge},j,k), \\ & n(i_{surcharge},j,k) = \frac{1}{2} \cdot n(i_{surcharge},j,k) \end{array}$$

Cette méthode permet de rééquilibrer les charges entre les deux serveurs considérés.

Mais de cette façon nous ne tenons pas en compte les différentes charges des deux serveurs considérés, ni du volume de calcul dont est responsable la donnée séléctionnée pour réplication. Pour cela, nous pourrions utiliser les mêmes principes que dans les heuristiques **MaxOverMedianServ** et **MinOverMedianServ** et nous servir des volumes de calcul, en plus ou en moins, nécessaire pour atteindre le temps moyen de traitement des requêtes en attente dans les files des serveurs. Nous pouvons également nous servir du volume de calcul dont est responsable la donnée séléctionnée sur le serveur surchargé. L'idée de l'heuristique **MedianCompSched** est de redistribuer autant de calcul possible au serveur $P_{i_{sous-charge}}$ issue de la donnée $d_j$ pour que $P_{i_{surcharge}}$ ou $P_{isous-charge}$ atteigne la moyenne. Ainsi, la fraction $x$ des requêtes concernant $d_j$ que $P_{i_{surcharge}}$ cèdera à $P_{i_{sous-charge}}$ sera égale à :

$$x = min(1, \frac{(T_{i_{sous-charge}} - T_{moyen}) \cdot w_{i_{sous-charge}}}{w(i_{surcharge},j)}, \frac{(T_{i_{surcharge}} - T_{moyen}) \cdot w_{i_{surcharge}}}{w(i_{surcharge},j)})$$

où $T_i$ et $w(i,j)$ sont définis tels que précédemment. Les nouvelles valeurs d'ordonnancement concernant la donnée $d_j$ pour les deux serveurs concernées sont alors :

$$\begin{array}{lrcl} \forall k, v(j,k) \neq 0 : & n(i_{sous-charge},j,k) & = & n(i_{sous-charge},j,k) + x \cdot n(i_{surcharge},j,k), \\ & n(i_{surcharge},j,k) & = & (1-x) \cdot n(i_{surcharge},j,k) \end{array}$$

Afin de trouver le meilleur ordonnancement possible tenant compte du nouveau placement, une possibilité est de réutiliser le programme linéaire défini dans la section 4.3. En fixant le placement des données telles qu'il est réellement, la solution de ce programme nous donne les informations qui conduiront à un ordonnancement optimal dans les conditions actuelles. Mais les réexécutions du programme sont dues à la nécessité de corriger un problème de surcharge d'un des nœuds et donc à une utilisation différente des données que celle initialement prévue. Dans ce cas, plutôt que de se servir des fréquences prises en compte lors du placement initial, il est plus judicieux d'utiliser les fréquences d'utilisation qui ont réellement été rencontrées jusque-là dans

le flot de requêtes. La dernière technique de réordonnancement, **LPSched** , utilise donc les fréquences d'utilisation du flot de requête réellement reçu par l'ordonnanceur et le placement existant des données afin de recalculer les informations d'ordonnancement optimale avec le programme linéaire de la section précédente.

Une fois que l'ordonnancement à été ajusté pour tenir compte des modifications de charge et de placement, la dernière étape consiste à resoumettre les jobs concernés par ces modifications qui se trouvent dans les files d'attente. C'est-à-dire que pour tous les nœuds dont au moins une des valeurs d'ordonnancement à été modifié il faut examiner la liste des requêtes qu'il a dans sa file. Pour chaque requête concernée par un changement, alors celle-ci sera extraite de la file d'attente et puis renvoyée au gestionnaire de requête afin d'être réordonnancée en conformité avec les nouvelles informations.

### 4.4.8 Résumé des différentes heuristiques

**Algorithme 4.3** Algorithme général d'adaptation dynamique de la charge

1: **Tant que** il y a des requêtes à traiter **:**
2: Regarder la charge des serveurs
3: **Si** il y des serveurs en surcharge **Alors**
4: Choisir le serveur $P_{surcharge}$ le plus surchargé
5: Déterminer la donnée $D_{charge}$ cause de la surcharge
6: Choisir un serveur $P_{sous-charge}$ en sous-charge
7: **Si** il y a assez de place pour répliquer $D_{charge}$ sur $P_{sous-charge}$ **Alors**
8: Repliquer
9: **Sinon**
10: **Tant que** il n'y a pas assez de place sur $P_{sous_charge}$ pour $D_{charge}$ **:**
11: Choisir une donnée à supprimer sur $D_{charge}$
12: **Si** il n'y a toujours pas assez de place **Alors**
13: Restaurer les données supprimées
14: Choisir un autre serveur pour $P_{sous_charge}$ et recommencer à partir de 7
15: **Si** des données ont été répliquées ou supprimées **Alors**
16: Mettre à jour l'ordonnancement
17: Resoumettre les requêtes concernées
18: Attendre un temps $delai$

L'algorithme 4.3 résume de façon générale les différentes étapes à effectuer pour que notre système puissent s'adapter de façon dynamique à des variations de charge provoqué par une utilisation non prévue de certaines données. Il s'agit là de la structure générale du fonctionnenement du module DynSRAManager chargé de l'évaluation des charges et de l'adapation du placement et de l'ordonnancement si nécessaire. Dans les étapes 4, 5, 6, 11 et 16 de cet algorithme, il faut effectuer des choix, de serveurs ou de données, dont la finalité est de rétablir l'équilibre global de charge entre

les nœuds du système. Nous avons défini précédemment plusieurs métriques et heuristiques afin d'essayer de faire des choix pertinents. Refaisons maintenant un rapide résumé de ces méthodes :

**Etape 4** : choix du serveur en surcharge.
- **MaxCompServ** : le serveur le plus chargé en volume de calcul
- **MaxTimeServ** : le serveur le plus chargé (en temps)
- **MaxOverMedianServ** : le serveur qui nécessite le plus de volume de calcul pour atteindre la moyenne.

**Etape 5** : choix de la donnée dont on va chercher à répartir les requêtes sur un autre serveur.
- **MaxCompData** : la donnée la plus coûteuse en volume de calcul
- **MaxDiffData** : la donnée dont l'utilisation ne correspond pas avec les prévisions initiales

**Etape 6** : choix du serveur en sous-charge.
- **MinCompServ** : le serveur le moins chargé en volume de calcul
- **MinTimeServ** : le serveur le moins chargé en temps de calcul
- **MaxUnderMedianServ** : le serveur le plus en dessous de la moyenne en volume de calcul

**Etape 11** : choix de la donnée à supprimer.
- **LessUsedData** : donnée la moins utilisée dans l'historique
- **MinCompData** : donnée la moins utilisée en volume de calcul

**Etape 16** : méthode d'adaptation de l'ordonnancement.
- **LPSched** : calcul du programme linéaire en fixant le nouveau placement
- **EqShareSched** : partage égale entre le nouveau serveur et l'ancien
- **MedianCompSched** : partage entre les serveurs en fonction de leurs charges

## 4.5 Conclusion

Dans ce chapitre, nous avons commencé par poser une modélisation de la plate-forme de calcul qui nous intéresse ainsi que des différents paramètres permettant de décrire les caractéristiques des données et des applications bioinformatiques ainsi que leur utilisation. À partir de cela, nous avons établi un programme linéaire permettant de calculer un placement statique des données ainsi que des informations sur l'ordonnancement des requêtes en fonction de ce placement offrant un rendement optimale pour la plate-forme. Nous avons montré que ce le problème lié à l'ordonnancement

est un problème NP-complet et nous avons alors proposé une heuristique dont le but est d'approcher la solution optimale par approximation successives. Nous avons également expliciter un algorithme glouton qui résout le problème du placement dont le but est de saturer l'espace de stockage en essayant de répliquer d'abord les banques de données les plus consommatrices en calculs.

Dans un second temps, nous avons considéré le fait que, suite à des différences d'utilisation des données par rapport aux prévisions, d'importants déséquilibres de charges pouvaient apparaître entre les nœuds. De tels déséquilibres auraient pour effet une utilisation non efficace de la grille dûe à un placement et un ordonnancement mal adaptés aux conditions d'utilisations des données actuelles. Dans ce cas, nous avons proposé différentes métriques pour évaluer ces déséquilibres entre les nœuds afin d'être en mesure de déterminer quels nœuds et quelles données nécessitent une attention particulière afin de pouvoir rétablir l'équilibre de charge entre les nœuds et améliorer le rendement de la plate-forme. Ces différentes heuristiques et métriques combinées entre elles avec une surveillance régulière de la charge des nœuds de calculs nous permettent d'adapter le placement et l'ordonnancement des données et être ainsi capable de répondre à des modifications dans les schémas d'utilisation des données.

# Chapitre 5

# Évaluation

## 5.1 Introduction

Les plates-formes cibles sur lesquelles portent notre étude sont des environnements largement distribués qui peuvent être situés sur de nombreux domaines administratifs. Par conséquent, il peut être relativement difficile de conduire des expériences reproductibles de longue durée dans un tel cadre. Nous avons donc choisi de faire usage d'un simulateur afin de tester nos algorithmes. Un autre avantage de l'utilisation d'un simulateur est la possibilité de pouvoir maîtriser l'ensemble des paramètres de la plate-forme simulée, ce qui est peut être difficile voir impossible dans un environnement réel. Notre choix s'est porté sur OptorSim [1], un simulateur développé dans le cadre du projet européen DataGrid [40]. Il s'agit d'un simulateur conçu pour tester différents algorithmes de réplication et de gestion de données au sein de la grille DataGrid. L'objectif initial de sa conception s'avère donc assez proche du notre, et de ce fait, ce simulateur intègre de nombreuses fonctionnalités dont nous avons besoin comme les mécanismes nécessaires à la simulation de la gestion et de la localisation des données. Ces mécanismes indispensables pour nos travaux auraient dus être complètement réécrits dans un simulateur tel que SimGrid [10] par exemple.

Dans ce chapitre nous allons présenter le simulateur utilisé avec toutes les modifications que nous y avons apportées afin de correspondre à nos besoins. Nous verrons également les différents paramètres qui ont été utilisés pour configurer le simulateur lors des expériences. Enfin, et surtout, nous verrons les résultats obtenus par les différents algorithmes présentés dans le chapitre précédent.

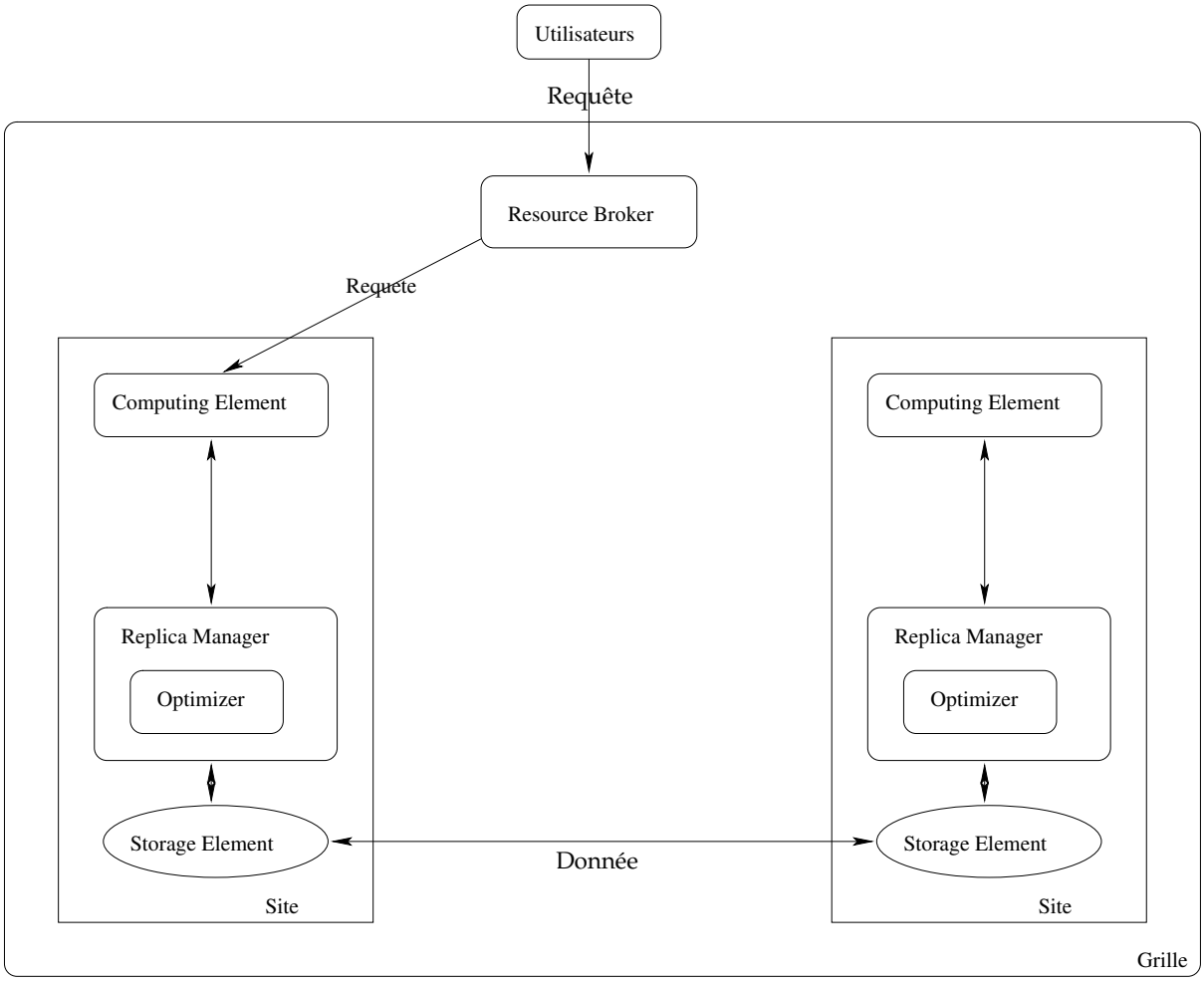


FIG. 5.1 – Architecture fonctionnelle du fonctionnement d'OptorSim. On y retrouve les différents éléments qui constituent la grille simulée.

## 5.2 Le simulateur de grilles de données OptorSim

### 5.2.1 Fonctionnement global

La figure 5.1 montre les principaux éléments qui constitue la grille simulée. Cette grille est composée de *sites*. Un site contient une unité de calcul appelé *Computing Element (CE)*, un gestionnaire de réplicats, le *Replica Manager (RM)* ainsi qu'une unité de stockage de données, le *Storage Element(SE)*. En dehors de ces sites se trouvent deux autres entités : l'*Utilisateur*, qui génère les requêtes et les soumet à l'ordonnanceur de la grille, *le Resource Broker(RB)*.

Détaillons un peu le fonctionnement de ces différents éléments.

**Computing Element (CE) :** il simule une machine seule ou une grappe de machines. Il a pour rôle de simuler l'exécution des requêtes qui lui sont soumises. Lorsqu'un CE s'apprête à exécuter une requête il s'adresse au Replica Manager afin de récupérer les différentes données nécessaires à l'exécution. Un CE est défini par le nombre de nœuds qui le compose. La durée d'exécution des requêtes est identique pour toutes les requêtes quelque soit leur type, les données auxquelles elle accède ou le CE sur lequel elle est exécutée. Cette durée est fixée dans les fichiers de configuration du simulateur. Lors de la simulation de l'exécution d'une requête, ce temps fixe est divisé par le nombre de nœuds du CE.

**Replica Manager (RM) :** il y en a un par site et il a en charge la localisation et la gestion des mouvements des données. Le RM contient un *Optimizer*, il s'agit d'un objet dans lequel sont définies les stratégies de réplication et de suppression de données. Ce que les auteurs d'OptorSim appellent stratégies de réplication sont en fait la façon dont sont calculés les coûts de transfert d'une donnée, la façon dont est choisie la source d'un transfert ainsi que le choix des données à supprimer sur le SE lorsque cela est nécessaire. Il s'agit donc principalement de stratégies locales au niveau d'un site.

**Storage Element (SE)** : il a la charge du stockage et est caractérisé par sa capacité de stockage (en Mégaotets). Un même fichier peut être stocké sur différents SE au même moment, et chaque SE a la charge de décider quel fichier doit être supprimé si de l'espace est requis pour un nouveau fichier alors que le SE est déjà plein. Comme cette décision est locale et ne se fait pas en concertation avec les autres SE ni le gestionnaire de réplicats, il est possible que toutes les copies d'un même fichier soient supprimées simultanément et que ce fichier ne soit alors plus accessible dans la grille. Pour pallier à ce problème, pour chaque fichier, il est possible de définir au moins une copie maître qui ne peut être supprimée du serveur où elle se trouve. La façon dont les copies maîtres sont distribuées est définie par une option de configuration. Les copies maîtres peuvent être placées sur un serveur ou bien répartie aléatoirement sur une liste de serveurs identifiés dans le fichier de configuration. Cette distribution est faite à l'initialisation de la simulation, ainsi le coût pour placer ces copies maîtres est considéré comme nul. Pour fonctionner correctement un CE doit avoir un SE local. Par contre, un SE peut très bien exister sans être associé à un CE, il peut être vu alors comme un dépôt de données distant. Le temps d'accès aux données situées sur le SE local par le CE est considéré comme négligeable.

**Utilisateur** : il est la source des requêtes. C'est lui qui crée les requêtes selon un schéma bien précis. La encore, divers schémas sont fournis et le choix de celui qui sera utilisé est fait dans la configuration.

**Resource Broker (RB)** : c'est l'organe central de décision. Il est capable de communiquer avec les différents CEs et RMs pour récupérer toutes les informations né-

cessaires à l'ordonnancement. Plusieurs algorithmes sont disponibles pour effectuer l'ordonnancement, le choix de celui qui doit être employé est défini dans les fichiers de configuration du système.

### 5.2.2 Adaptation du simulateur

#### 5.2.2.1 Modification de l'existant

En l'état, le simulateur ne nous permettait pas d'effectuer exactement les expériences qui nous intéressaient. Si nous n'avons que peu modifié le fonctionnement interne du simulateur, nous l'avons largement étendu et complété afin de le rendre plus souple et plus en adéquation avec notre modèle.

**Simulation de l'exécution** Le temps d'exécution des requêtes ne faisait pas parti des points auxquels s'intéressaient les développeurs d'OptorSim. Le temps de calcul est un temps fixe peut importe la requête et le nombre de données auxquelles elle a besoin d'accéder, seuls les temps de transferts de ces données sont réellement pris en compte. Ce modèle n'était pas cohérent avec notre modèlisation théorique.

Nous avons donc rajouté, pour chaque CE, la possibilité de fixer une capacité de calcul exprimée en flops. De plus nous avons permis de définir, pour chaque type de requête, un temps d'exécution affine en la taille des données utilisées.

Dans la version officielle d'Optorsim, il est possible de définir pour chaque CE un nombre de nœuds de calcul, pour simuler l'utilisation de grappes. Mais l'utilisation qui en était faite n'était que peu réaliste. Nous avons donc implémenté le CE de façon à simuler un gestionnaire de batch en rajoutant un élément supplémentaire les *Worker Node (WN)* qui exécuteront les tâches. Le CE, quant à lui, agit comme le maître d'un tel système et a la charge de recevoir les requêtes du Resource Broker et de les fournir à ses Worker Node. Ainsi lorsqu'une requête est envoyée à un CE par le Resource Broker, celle-ci est placée dans une file d'attente. Cette file est gérée de façon FIFO, c'est-à-dire que les premières requêtes qui en sortiront seront les premières requêtes à y être rentrées. À chaque fois qu'un WN termine le traitement d'une requête, il accède à la file du CE auquel il est rattaché. Si une requête s'y trouve, alors il la prend pour lui, la retire de la file et effectue les traitements liés à la requête. Bien évidemment, les accès à cette file se font en exclusion mutuelle afin que deux WNs ne puissent s'attribuer et traiter la même requête.

La capacité de calcul définie pour les CE est alors en fait celle des Worker Node. Tous les WN d'un même CE ont la même puissance de calcul. La grille simulée est alors un ensemble hétérogène de grappe homogène.

**Création des requêtes** La méthode de création des requêtes n'était pas assez souple pour nos besoins. En effet, les requêtes étaient crées aléatoirement en suivant une pro-

babilité fixée au départ et les fichiers accédés par une requête étaient déterminés au moment de l'exécution sans contrôle vraiment précis. Au vu des connaissance que nous avons des requêtes pour lesquelles nos algorithmes sont définis, il nous fallait plus de contrôle sur le processus de création. Dans notre version, les requêtes sont crées en suivant une liste de couples (algorithme - banque de données) décrite dans un fichier de configuration passé en paramètre au simulateur. De cette façon, la liste des requêtes soumises au cours de la simulation est totalement définie avant le démarrage de la simulation. Cela permettra, entre autres, de se servir de cette liste comme source pour définir les fréquences d'utilisation des algorithmes et des banques de données. Cela offre également la possibilité de rejouer plusieurs fois des simulations avec le même jeu de requêtes.

**Ordonnancement** L'ordonnanceur à également été largement modifié. Dans la version initiale du simulateur, l'ordonnancement se fait sur des variantes de *MCT* (*Minimum Completion Time*), c'est-à-dire un ordonnancement dynamique qui prend en compte les différents paramètres de la plate-forme au moment de l'ordonnancement pour envoyer la requête au serveur qui devrait pouvoir le terminer le plus tôt. Dans notre cas, l'ordonnancement est effectué à partir d'informations calculées à l'avance. Une nouvelle entité est donc apparue au sein du Resource Broker, dont le rôle est de regrouper les informations d'ordonnancement déterminées par nos algorithmes et de permettre à l'ordonnanceur de les utiliser pour décider sur quel serveur envoyer chaque requêtes. Pour des raisons de comparaison avec nos algorithmes avec d'autres méthodes, nous avons conservé MCT. Mais puisque nous avons apporté beaucoup de modifications sur ce que le système est capable de connaitre au sujet des requêtes, nous avons choisi d'enrichir MCT afin de tenir compte de ces nouvelles informations. Ainsi, dans la version originale d'OptorSim, l'ordonnancemnet MCT se fait principalement en regardant les temps d'accès nécessaire pour accéder aux données requises par les requêtes en attente dans la file de chaque CE, plus le temps d'accès aux données de la requête en cours d'exécution. Notre version du MCT prend en considération un peu plus d'éléments. Bien evidémment les temps d'accès sont toujours pris en compte mais nous y avons rajouté le temps d'exécution de l'ensemble des requêtes dans la file d'attente. En effet, puisque nous avons défini un modèle pour ces temps d'exécution, il est facile d'en avoir une bonne estimation pour un ensemble de requêtes.

#### 5.2.2.2 Rajout

La version dynamique de notre algorithme nécessite de recalculer le placement et l'ordonnancement au cours de l'exécution. Nous avons donc rajouté un module supplémentaire au schéma de base. Ce module a été nommé DynSRAManager et a été conçu pour que l'on puisse facilement y implanter les différentes heuristiques liées à ces algorithmes.

Ce module a pour charge de contacter régulièrement l'ensemble des CEs afin de connaître leur charge actuelle. À partir de ces informations, et en fonction des heu-

ristiques en cours d'expérimentations, il va, éventuellement, déterminer un nouveau placement pour les données ainsi que de nouvelles directives d'ordonnancement prenant ce placement en compte. Après calculs, il informera l'ordonnanceur des nouvelles orientations de l'ordonnancement et le gestionnaire de réplicat du nouveau placement.

La figure 5.2 représente l'architecture globale du simulateur après les modifications que nous y avons apportées. Si nous comparons avec la figure 5.1 représentant la structure initiale d'Optorsim, nous pouvons constater que toute la structure principale a été conservée, mais que le fonctionnement de certains éléments, comme les CE par exemple, a été plus détaillé afin d'être plus réaliste. D'autres éléments ont été rajoutés ou modifiés afin de permettre la mise en œuvre de nos algorithmes.

En terme de lignes de code, l'ensemble des rajouts et modifications apportés au code originel du simulateur représente environ $8000$ lignes de codes dont $6000$ de rajoutés soit environ 45% de la taille du code d'origine. Si l'on rajoute, le code contenant les différents algorithmes et heuristiques, on atteint un total de $12000$ lignes de codes supplémentaires. Après de nombreuses discussions avec l'équipe de développement du simulateur Optorsim, il a été décidé qu'une grande partie de ces modifications, principalement celles concernant le CE et les WN, devraient apparaître dans une future version du simulateur.

## 5.3 Environnement et paramètres de simulation

### 5.3.1 La plate-forme

Comme nous venons de le voir, notre outil permet de simuler une grille hétérogène de grappes homogènes. Mais la topologie de ces grilles est un point très important. En effet, la topologie sous-jacente peut avoir un impact significatif sur les performances des heuristiques étudiées. Nous avons donc utilisé le logiciel Tiers [23] pour générer un ensemble de topologies. Le logiciel Tiers est largement reconnu pour être un générateur de topologie de réseau réaliste. Tiers génère aléatoirement des topologies réseaux caractéristiques de l'Internet. Ces topologies ont trois niveaux de hiérarchie : WAN, MAN et LAN. Tiers commence par placer les nœuds WAN aléatoirement dans un espace géométrique euclidien et crée un arbre de recouvrement minimal entre ces nœuds. Ensuite, il ajoute des arêtes entre les nœuds de façon congruente avec une loi-puissance (power law) sur le degré des nœuds observé sur l'Internet. Les réseaux MAN et WAN sont construits suivant la même méthode. Enfin, les trois types de réseaux sont connectés ensemble. La figure 5.3 montre un exemple d'architecture générée par Tiers.

Tiers permet de construire la plate-forme de base avec les nœuds et les connections. Mais cette plate-forme n'est pas exactement celle que nous désirons avoir, puisque ce qui nous interresse, c'est une collection hétérogène de grappes homogènes. Dans notre processus de génération de plate-forme, nous remplacerons donc tous les nœuds

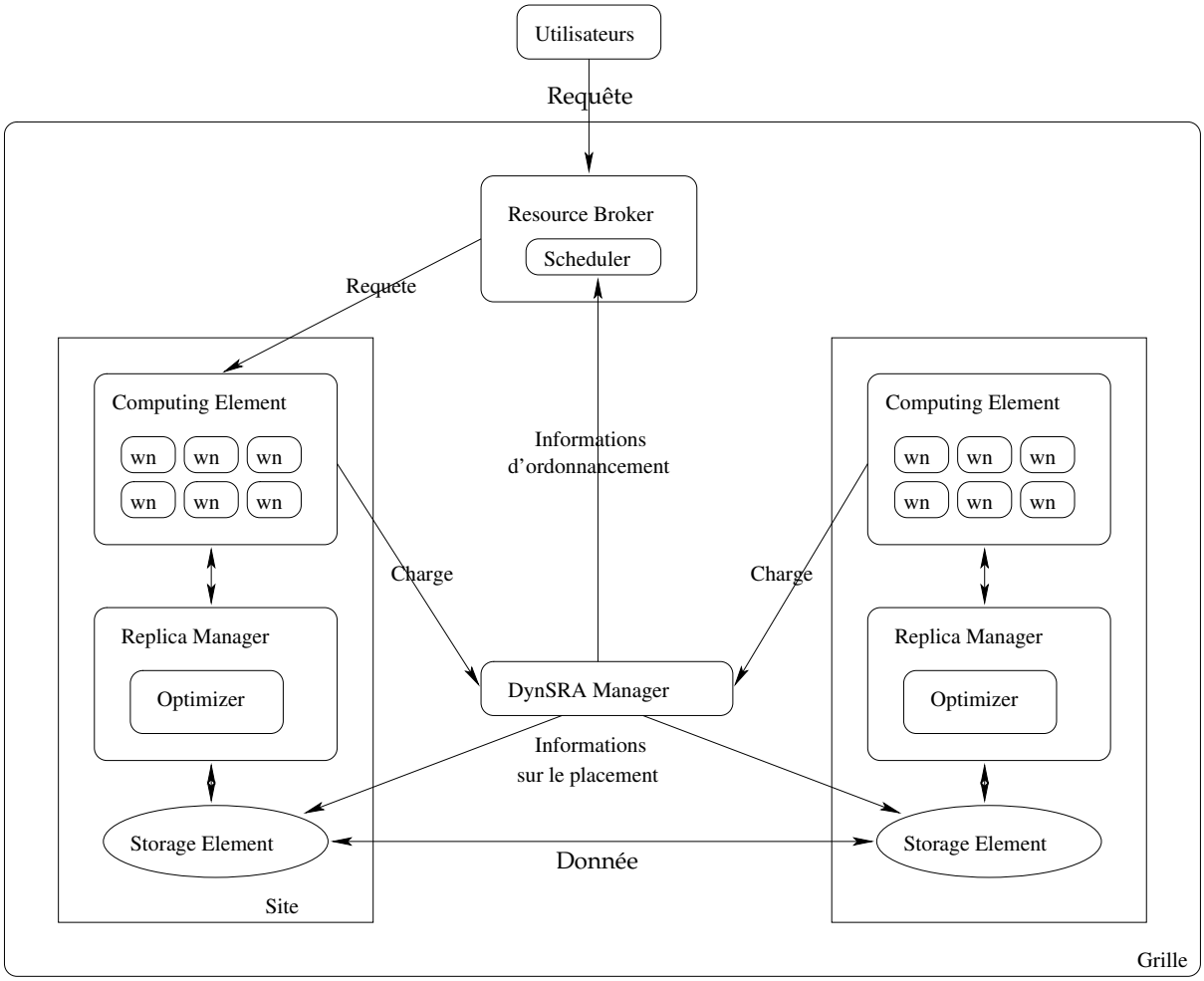


FIG. 5.2 – Architecture fonctionnelle du fonctionnement de notre simulateur basé sur Optorsim.

situés en LAN par des grappes de machines. Les nœuds des zones WAN et MAN seront vus comme des routeurs.

Tiers propose également d'affecter des débits de données aux liens entre les nœuds. Les paramètres par défaut sont comme indiqué dans la table 5.1. Mais en fonction des expériences ces paramètres ont souvent dû être modifiés pour voir l'impact du débit des réseaux sur nos différents algorithmes. Nous détaillerons ces modifications lors de la description des expériences.

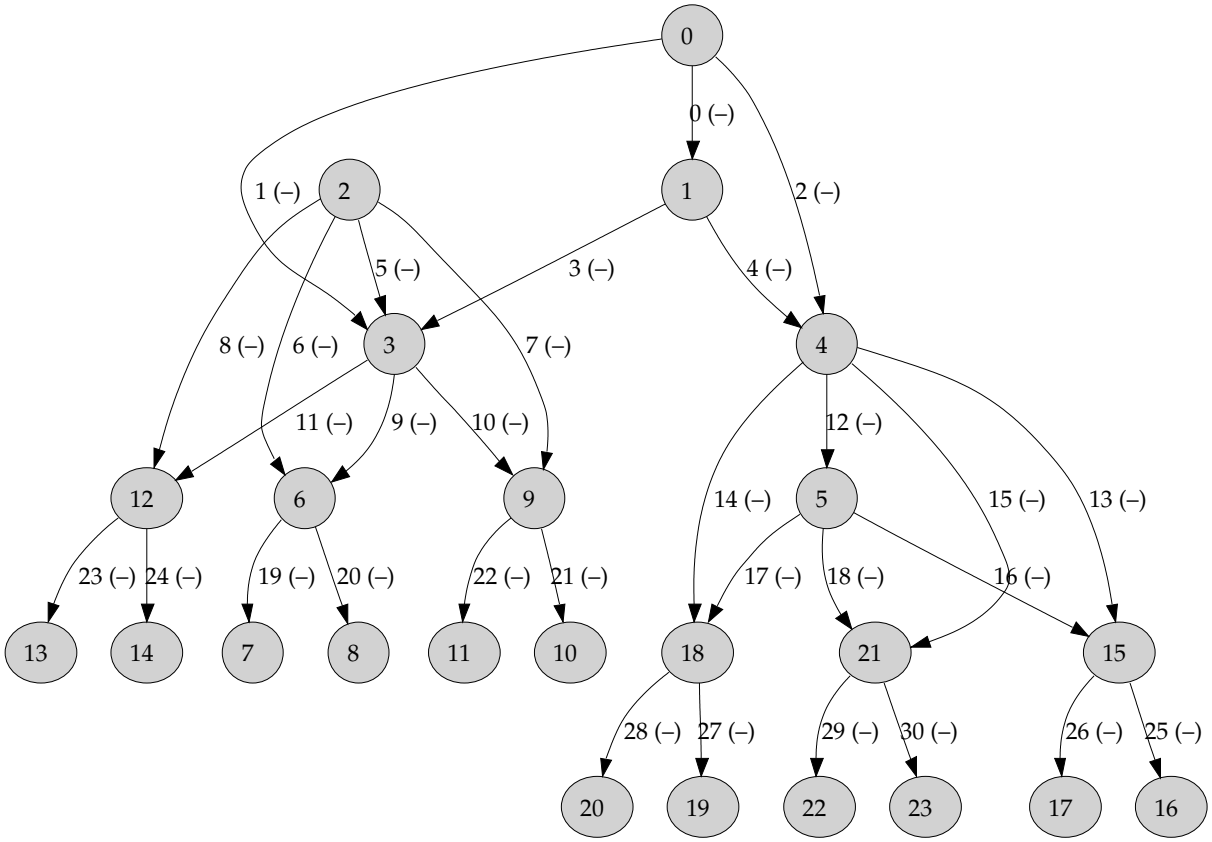


FIG. 5.3 – Exemple d'architecture de réseaux générée par Tiers. Les nœuds 0 et 1 sont des nœuds du WAN, de 2 à 5 les nœuds sont dans le MAN et de 6 à 23 les nœuds sont dans la zone LAN.

| | WAN | MAN | LAN |
|---|---|---|---|
| WAN | 10 Mb/s | 10 Mb/s | |
| MAN | 10 Mb/s | 100 Mb/s | 100 Mb/s |
| LAN | | 100 Mb/s | 1000 Mb/s |

TAB. 5.1 – Débits des liens entre les nœuds suggérés par Tiers.

### 5.3.2 Les paramètres d'entrées

Que ce soit pour la plate-forme de simulation ou les données nécessaires à l'exécution de nos algorithmes, de nombreux paramètres sont nécessaires. Si l'on regarde les algorithmes, les paramètres dont ils ont besoin pour fonctionner sont à peu près toujours identiques :

- une description complète de la plate-forme d'exécution,
- la fréquence d'apparition des requêtes,

– une estimation du temps de calcul pour chaque type de requêtes,
– enfin une description des données utilisées.

Nous avons déjà vu que la topologie de la plate-forme d'exécution était définie par Tiers. Les paramètres que nous avons fournis ont permis de générer des plate-formes ayant entre 10 et 15 grappes réparties en environ 6 à 7 MAN. Les grappes de calculs sont composées de 2 à 64 nœuds. La puissance des nœuds de chaque grappe est identique mais les grappes ayant un nombre different de nœuds, nous avons bien de l'hétérogénéité au sein de la grille.

Les fréquences d'apparition des requêtes sont établies à partir des traces que nous avons obtenu du serveur NPS@. Les traces que nous possédions n'intégraient pas toutes les bases existentes, et forcement, toutes celles à venir. Pour nos travaux, nous avons donc choisi d'augmenter artificiellement la taille et le nombre des banques de données. Nous avons toutefois cherché à conserver un certain réalisme dans la distribution de l'ensemble des tailles des banques de données.

## 5.4 L'approche statique

### 5.4.1 Bref rappel de la méthode

Il s'agit, dans cette approche, de calculer une placement statique des données ainsi que des informations d'ordonnancement des requêtes sur ces données. Ces calculs sont réalisés par l'entremise d'un programme linéaire prenant en compte les différents paramètres de la plate-forme de calcul ainsi que les connaissances sur la fréquence des requêtes.

### 5.4.2 Méthode d'évaluation

Nous avons réalisé des simulations avec trois types de placements et ordonnanceurs afin de comparer les résultats de notre méthode. Dans la suite, *SRA* (Schedule Replication Algorithm) désignera notre stratégie. L'ordonnancement et le placement des simulations de ce type sont réalisés avec la solution entière du programme linéaire. Pour *MCT* [1], seul le placement suit le résultat de notre programme linéaire. L'ordonnanceur est de type online : à l'arrivée d'une requête, il essaie d'évaluer quel serveur de calcul est capable de finir le calcul en premier en tenant compte de la puissance des serveurs, des tâches qu'ils ont en attente et en cours d'exécution ainsi que de l'éventuel temps de transfert des données. *Greedy* est le résultat obtenu avec le placement issu de l'algorithme glouton 4.2. L'ordonnancement est effectué comme pour *MCT*. Les simulations sont réalisées pour un ensemble de 40000 requêtes. Les temps de calcul sont normalisés par rapport à la valeur théorique optimal du programme linéaire en rationnel. C'est-à-dire que la durée nécessaire pour exécuter l'ensemble des

[1] *Minimum Completion Time*

requêtes au cours de la simulation est divisé par le temps qu'il aurait fallu pour traiter un flot de requête ayant les mêmes caractéristiques s'il avait été possible de suivre exactement la solution rationnel du programme, i.e. avant l'approximation entière, du programme linéaire. La taille de l'espace de stockage disponible sur l'ensemble de la grille est, quand à elle, représentée par le ratio entre la somme des espaces de stockage disponible sur chacunes des grappes et la somme des tailles de l'ensemble des banques de données. Ainsi, une valeur de 10 pour ce ratio signifie que, globalement, il serait possible de stocker 10 réplicats de chacune des banques dans la plate-forme.

### 5.4.3 Résultats

La figure 5.4 montre le temps d'exécution de l'ensemble des requêtes soumises en fonction du débit du réseau connectant les nœuds entre eux. Dans cette simulation, le débit entre les nœuds est homogène. Ce choix est fait afin de mieux voir l'influence du débit sur les temps d'exécution. La figure 5.5 représente le volume de données transféré au cours de la même simulation, en fonction du débit du réseau. Nous pouvons voir que dans le cas de SRA et de greedy, le temps d'exécution est constant et indépendant du débit du réseau. Cela s'explique par le fait que, dans ces deux cas, aucun mouvement de données n'est effectué durant l'exécution comme nous pouvons le constater sur la Figure 5.5.

Mais les raisons de l'absence de mouvement de données ne sont pas les mêmes dans les deux cas. Dans le cas de SRA, l'ordonnancement est fait selon les résultats du programme linéaire qui calcule en même temps le placement des données. L'ordonnanceur envoie donc des requêtes uniquement sur des serveurs qui possèdent les données nécessaires à son exécution. Pour greedy, cela est dû au fait que l'algorithme glouton sature totalement l'espace de stockage de l'ensemble des serveurs en privilégiant les données les plus consommatrices en termes de ressources de calculs. Ainsi, lors de l'ordonnancement, l'algorithme envoie systématiquement les calculs vers les serveurs qui ont déjà les données requises. MCT quant à lui effectue un grand nombre de transfert de donnée en privilégiant le temps d'exécution de la requête en cours d'ordonnancement. Mais son absence de connaissance du schéma d'utilisation des requêtes fait que, globalement, il effectue un grand nombre de transferts inutiles. De la sorte, il ne devient très efficace que lorsque le débit du réseau est très important et donc que les temps de transfert deviennent négligeables par rapport à l'ensemble des temps de calculs.

La figure 5.6 montre le temps d'exécution du même ensemble de 40000 requêtes en fonction de l'espace de stockage total disponible sur la plate-forme considérée. L'espace est exprimé sous la forme d'un ratio entre l'espace de stockage total disponible sur la plate-forme et la somme de taille des banques de données utilisées. Pour cette simulation, le débit du réseau a été fixé à 10 Mo/s, c'est-à-dire une valeur où, selon les simulations précédentes, notre algorithme est plus performant que les autres.

Nous pouvons remarquer que pour les trois types d'expérimentations, le temps

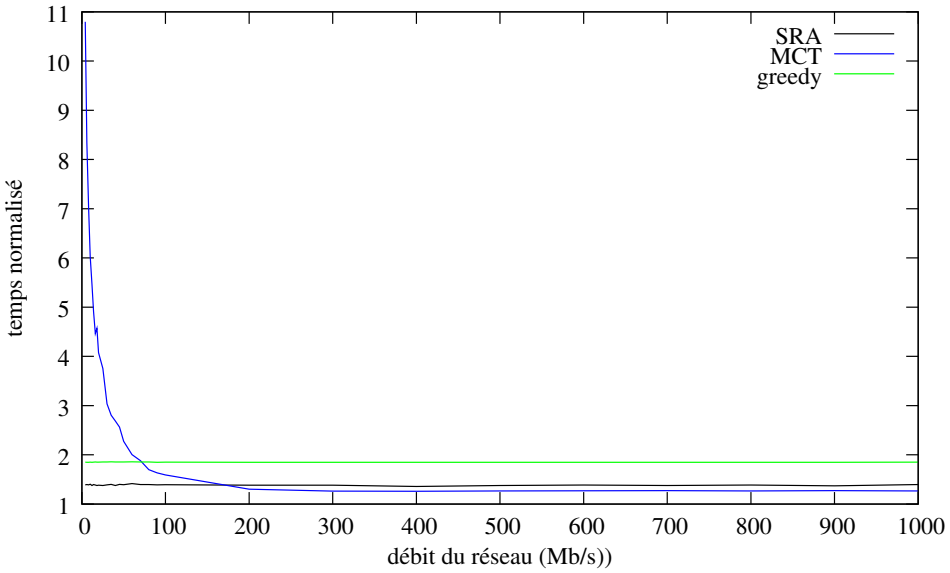


FIG. 5.4 – Temps d'exécution pour 40000 tâches en fonction de la bande-passante réseau entre les CE pour différents algorithmes de placement et d'ordonnancement.

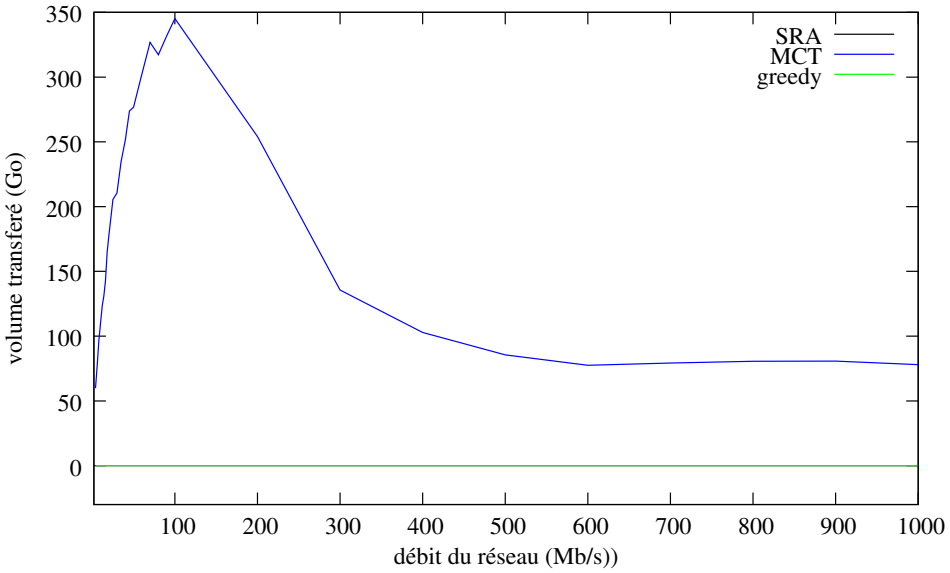


FIG. 5.5 – Volume de données transférées durant l'exécution.

d'exécution décroît à mesure que l'espace de stockage augmente jusqu'à atteindre une valeur constante. Cela s'explique logiquement par le fait que, plus il y a d'espace disponible, plus il est possible de placer des réplicats dans le système jusqu'à atteindre une valeur limite où le rendement maximal de la plate-forme en terme de calcul est atteinte. Lorsque l'espace de stockage est faible (inférieur à huit fois la taille de l'en-

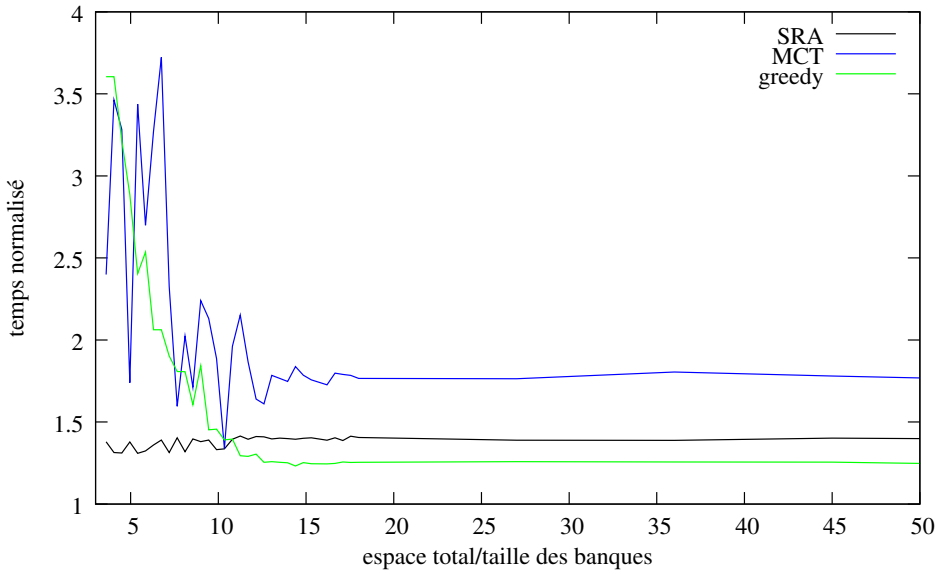


FIG. 5.6 – Temps d'exécution en fonction de la taille de stockage disponible.

semble des bases de données dans ce cas ci), notre algorithme obtient de meilleurs résultats que les autres méthodes. En effet, le programme linéaire permet d'optimiser au mieux les ressources restreintes disponibles grâce a une meilleure connaissance de l'utilisation des données. Avec l'augmentation de l'espace de stockage, les performances de l'algorithme glouton s'améliore pour dépasser celle obtenue par le programme linéaire. Cela arrive lorsque presque toutes les bases de données peuvent être répliquées sur presque tous les serveurs.

La Figure 5.7 est un agrandissement de la courbe précédente pour les résultats SRA et glouton. Lorsque l'espace de stockage est très restreint, les résultats de notre algorithme ainsi que du glouton ne sont pas réguliers. Cela provient de l'heuristique qui permet de construire la solution entière du programme linéaire. Dans ces conditions limites, l'impact d'un mauvais choix de placement est très important sur la valeur de la fonction objective, et dans ce cas nous pouvons constater de grandes différences entre la valeur de la fonction objective de la résolution réelle et de l'approximation entière. Dès que l'espace de stockage devient assez grand, notre heuristique permet de conserver la valeur de la fonction objective entre l'approximation entière et la solution réelle.

### 5.4.4 Les algorithmes d'approximation

Dans le but d'obtenir une estimation sur la performance de notre algorithme d'approximation entière de la solution du programme linéaire, nous l'avons comparé à une méthode d'approximation aléatoire basée sur celle utilisée dans [15]. Ces deux algorithmes sont très proches, la différence principale tient dans la façon de choisir la

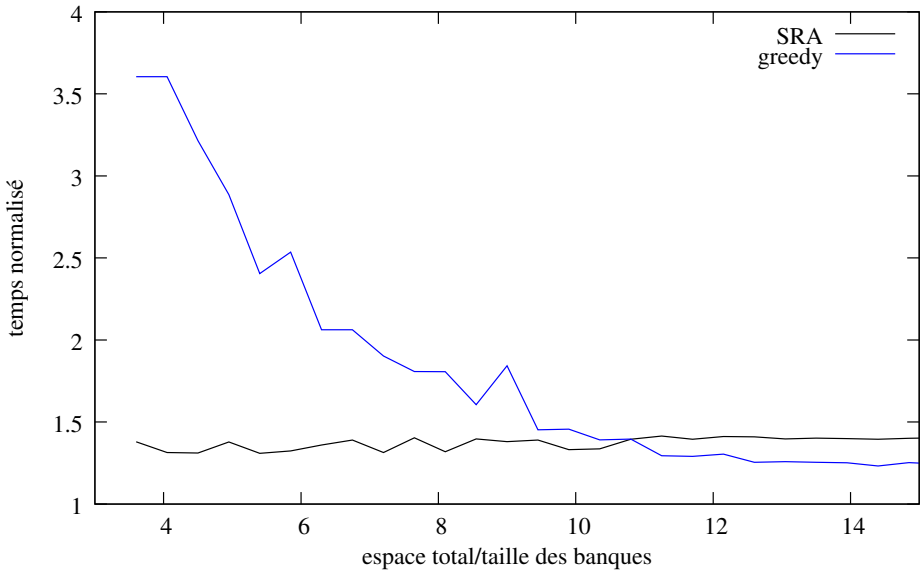


FIG. 5.7 – Zoom de la Figure 5.6.

variable dont la valeur sera approximée à chaque tour. Dans notre cas, cela est fait en prenant en compte les paramêtres du système et principalement la consommation en terme de volume de calcul des données. Dans le cas de l'approximation aléatoire, la variable est choisi aléatoirement parmi celles qui n'ont pas encore été fixées. Appelons $\delta_{i_1}^{j_1}$ une telle variable à fixer. Si la valeur courante dans la solution de programme linéaire de $\delta_{i_1}^{j_1}$ est supérieure à $0,5$ alors $\delta_{i_1}^{j_1}$ est fixé à $1$ sinon, il est fixé à $0$.

La figure 5.8 montre le temps d'exécution de $40000$ requêtes pour ces deux méthodes d'approximation. Comme nous pouvions nous y attendre, notre méthode d'approximation donne de bien meilleurs résultats . Cela s'explique évidemment par la façon dont le choix est fait qui permet de fixer à $1$ les données qui vont apporter le maximum de bénéfice dans le placement des données. Cette méthode permet, généralement, d'obtenir des solutions mixtes dont la valeur de l'optimal reste très proche de la valeur de l'optimal de la solution réelle.

## 5.5 L'approche dynamique

### 5.5.1 Bref rappel de la méthode

Nous allons maintenant utiliser le simulateur pour valider certaines des heuristiques liées à la version dynamique de notre méthode décrite dans le chapitre précédent. Cette méthode agit en plusieurs phases. Tout d'abord, il s'agit de déterminer si un ou plusieurs serveurs se trouvent en surcharge de calculs par rapport aux autres.

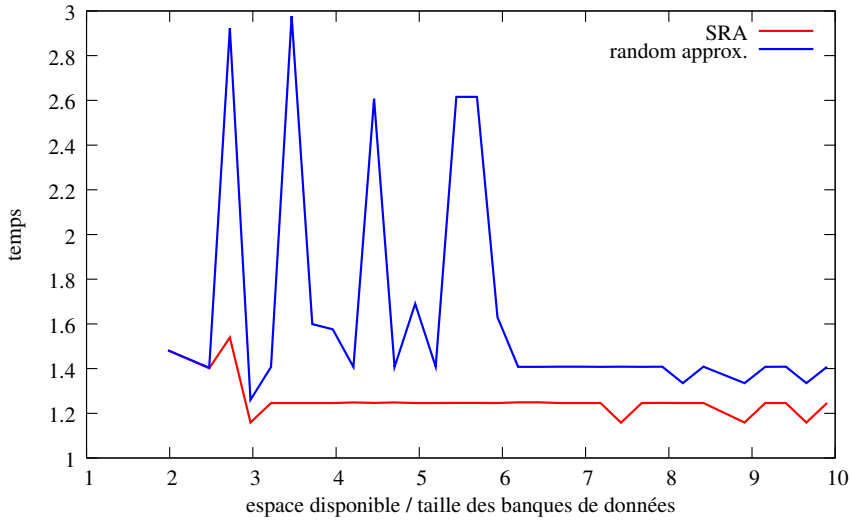


FIG. 5.8 – Temps d'exécution pour $40000$ requêtes en fonction de la taille de l'espace de stockage disponible pour deux techniques d'approximation.

Ensuite, il faut déterminer la donnée principalement responsable de la surcharge. Enfin, parmi les serveurs sous-chargés, il faut sélectionner un nœud qui pourra accepter une partie de la surcharge. Si nécessaire il faudra trouver des données à supprimer sur le serveur en sous-charge afin de répliquer la donnée. Pour chacune des ces étapes, nous avons établi plusieurs heuristiques que nous allons évaluer maintenant.

### 5.5.2 Création des requêtes

Notre version dynamique de SRA a été conçue pour être en mesure de répondre au cas où le flux de requêtes ne correspond pas à la distribution qui a servi à calculer le placement initial. Nous avons donc conçu deux méthodes afin de générer des ensembles de requêtes qui diffèrent de la distribution prévue de façon contrôlée. Dans la première méthode, l'ajustement des fréquences d'apparition des requêtes est faite par l'utilisation d'une valeur fixe $v$ pour tous les types de requêtes, mais l'ajustement peut se faire en plus ou en moins. L'établissement des fréquences se fait en commençant par compter le nombre $nb_{old}(j,k)$ de chaque type de requêtes $R(j,k)$ dans l'échantillon de référence. Ensuite, on calcule une nouvelle valeur $nb_{new}(j,k)$ telle que :

$$nb_{new}(j,k) = nb_{old} + nb_{old} \cdot \frac{v}{100} \tag{5.1}$$

Une fois que l'ensemble des nouvelles valeurs a été calculé, on peut établir les nouvelles fréquences $f(j,k)$ :

$$f(j,k) = \frac{nb_{new}(j,k)}{\sum_i \sum_k nb_{new}(j,k)} \tag{5.2}$$

La seconde méthode consiste à ne modifier la distribution que pour certains types de requêtes. Le principe de la modification est le même que dans la méthode précédente mais l'ajustement ne se fait pas pour tous les types de requêtes. Les types de requêtes dont l'utilisation vont varier sont choisis en fonction de leurs fréquences d'apparition dans la distribution initiale. Nous avons choisi d'effectuer ces modifications sur deux classes de requêtes. Le premier type de requêtes est celui correspondant aux dix requêtes les plus fréquentes dans l'ensemble initial. La deuxième classe est celle des dix requêtes les moins utilisées.

De cette façon, à partir d'un ensemble de requêtes, nous avons pouvons obtenir trois nouveaux jeux de fréquences. Le premier consiste en la modification de toutes les fréquences suivant un paramètre défini. Le second correspond aux tests où seules les fréquences d'utilisation des dix couples les plus présents dans la distribution initiale ont été modifiées. Enfin, dans le troisième jeu de test se sont les fréquences des requêtes les moins utilisées qui ont été modifiées. À partir de ces fréquences, il est facile de créer des ensembles de requêtes dont les fréquences d'apparition de chaque type correspond aux valeurs nouvellement créees et dont la différence par rapport à l'échantillon de requêtes initial peut être quantifiée par le facteur $v$ qui apparaît dans l'équation 5.1, appelé facteur de variation.

### 5.5.3 Les heuristiques

Dans la section 4.4, nous avons décrit différentes heuristiques permettant de choisir les différents paramètres intervenant dans la redistribution dynamique des données et des calculs dans le cas où des déséquilibres de charges entre les serveurs venaient à être détectés. Ces heuristiques portant sur des points différents de l'algorithme de redistribution, il y a plus d'une centaine de façons de les combiner entre elles. Nous avons fait le choix de ne pas simuler toutes ces possibilités et nous nous sommes restreints à un ensemble de 13 combinaisons qui sont résumées dans le tableau 5.2 à la page 77. Nous avons écarté toutes les combinaisons basées sur **MaxCompServ** et **MinCompServ** pour le choix des serveurs surchargés et en sous-charge. En effet, ces deux heuristiques ne prenant pas en compte la puissance de calcul des nœuds, leurs façons de mesurer la surcharge ne peut être fiable dans le cas de plates-formes hétérogènes en terme de puissance de calcul.

Les heuristiques **MaxTimeServ** et **MinTimeServ** de choix des serveurs en surcharge et sous-charge sont évidemment construites sur le même principe et utilisent le même critère de choix. Il en est de même pour les heuristiques **MaxOverMedianServ** et **MaxUnderMedianServ**. Nous avons décidé alors de ne pas « mixer » ces deux couples d'heuristiques. C'est-à-dire que nous avons uniquement considéré les combinaisons (**MaxTimeServ**-**MinTimeServ**) et (**MaxOverMedianServ**-**MinOverMedianServ**).

Par contre, afin d'établir l'utilité de la redistribution dynamique des données, nous avons également effectué deux types de simulations supplémentaires : la première

sans faire aucune redistribution, il s'agit donc du système SRA statique, et la seconde sans supprimer de données.

Dans toute la suite, lorsque nous ferons référence à un nombre en parlant d'une heuristique, il s'agira de la combinaison d'heuristiques portant ce numéro dans le tableau 5.2. Par exemple, l'heuristique 3 correspond à la combinaison des heuristiques **MaxTimeServ** et **MinTimeServ** pour la sélection des serveurs, **MaxCompData** et **MinCompData** pour le choix des données et **EquShareSched** pour le réordonnancement.

## 5.5.4 Résultats

### 5.5.4.1 Description des simulations

Toutes les combinaisons d'heuristiques présentées dans le tableau 5.2 ont été implémentées dans notre version du simulateur OptorSim et ont donné lieu à de nombreuses simulations en fonction des différents paramètres. Ces paramètres sont les mêmes que dans le cas de la version statique, à savoir le débit du réseau et le volume de stockage disponible dans la plate-forme. Il faut ajouter à ces paramètres celui induit par l'aspect dynamique de cette version qui est le facteur de variation entre les fréquences d'apparition des requêtes théoriques et réelles ainsi que les types de requêtes sur lesquels se portent ces variations.

Nous avons donc réalisé trois séries de tests en faisant varier à chaque fois l'un des paramètres parmi le facteur de variation, le volume de stockage ou le débit du réseau en fixant les autres paramètres. Pour chacune de ces trois séries nous avons créé trois ensembles de requêtes, l'un où toutes les fréquences d'apparitions sont modifiées, le second où seule les dix types de requêtes les plus fréquentes sont modifiés et le dernier où seulement les dix types de requêtes les moins fréquentes subissent des variations par rapport à l'échantillon initial. Ces ensembles de requêtes sont générés en suivant les méthodes décrites dans la section 5.5.2.

Chacun des résultats présentés dans le reste de cette section est en fait une moyenne des résultats obtenus sur 10 simulations effectuées avec des jeux de requêtes différents mais générés avec les mêmes paramètres.

### 5.5.4.2 En fonction de la variation

La première série de simulation est effectué en fixant les paramètres de stockage, et de réseaux pour l'ensemble des simulations à des valeurs moyennes. L'espace de stockage est répartie de façon homogène sur l'ensemble des nœuds et représente, au total, environ 10 fois la taille de l'ensemble des banques de données. Le réseau qui connecte les nœuds de la grille simulée suit les paramètres donnés par Tiers lors de la génération de la topologie. Pour chaque simulation, nous avons alors construit un jeu de requêtes en fixant le facteur de variation par rapport au jeu initial. Les figures 5.9

à 5.13 représentent le temps d'exécution d'une série de 40000 requêtes en fonction du facteur de variation entre la répartition des requêtes de l'échantillon originale et de l'ensemble réellement soumis au système. Le facteur de variation est exprimé en pourcentage, et le temps en centaines de secondes.

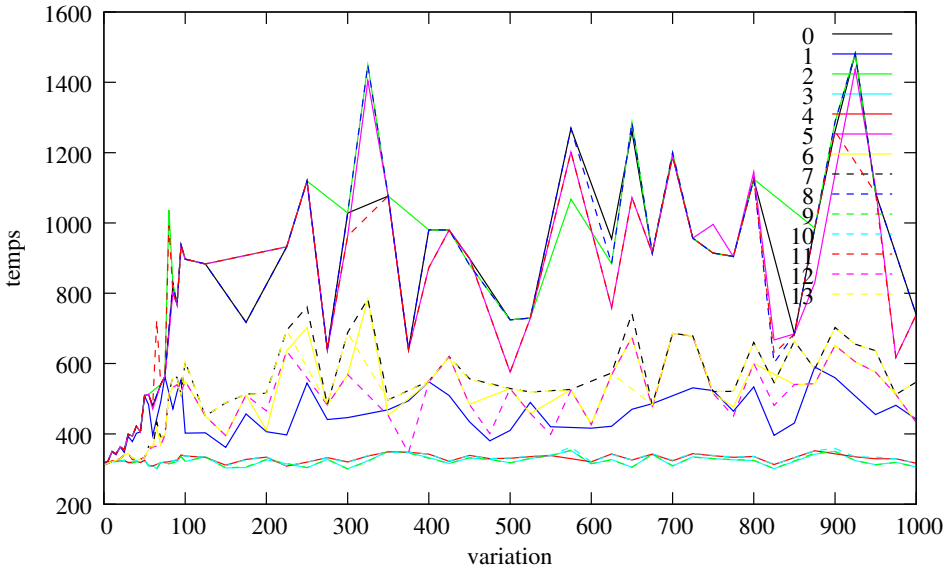


FIG. 5.9 – Temps d'exécution de 40000 requêtes en fonction de la variation entre les fréquences d'apparition de l'ensemble de requêtes initial et celles de l'ensemble des requêtes envoyées à l'ordonnanceur pour différentes heuristiques.

Les figures 5.9, 5.10 et 5.11 représentent les résultats obtenus lorsque les fréquences d'apparition de l'ensemble des requêtes sont modifiées. Sur la figure 5.9 nous distinguons trois ensembles dans lesquels les heuristiques ont des comportements similaires. Le nombre de courbes et leurs superpositions fréquentes rendent peu lisible ces graphiques. Comme nous pouvons voir que des heuristiques ont des comportements semblables, dans la suite, les graphiques ne représenteront plus toutes les courbes, seulement les enveloppes des courbes ayant un comportement proche. La figure 5.10 est une telle « factorisation » de courbes. Dans ce cas, la légende des courbes est alors la liste des heuristiques incluses dans cette zone. Cette figure permet de distinguer trois groupes bien distincts au sein desquels les heuristiques ont des performances similaires. Un groupe, constitué des heuristiques 3, 4, 9 et 10 se distingue particulièrement des deux autres, en ayant un temps d'exécution quasiment stable quelque soit la valeur du facteur de variation et toujours inférieur aux deux autres groupes. Ces simulations ont comme heuristiques communes **MaxCompData** et **MinCompData** et le réordonnancement est effectué soit par **EquShareSched** soit par **MedianCompSched** . Pour les deux autres groupes, le temps d'exécution augmente progressivement avec le facteur de variation entre 0 et 100. Une fois passé ce point les résultats montrent des temps d'exécution qui restent dans la même plage de valeurs. La figure 5.11 est un zoom de la figure 5.10 pour le facteur de variation entre 0 et 100. Lorsqu'il n'y a que

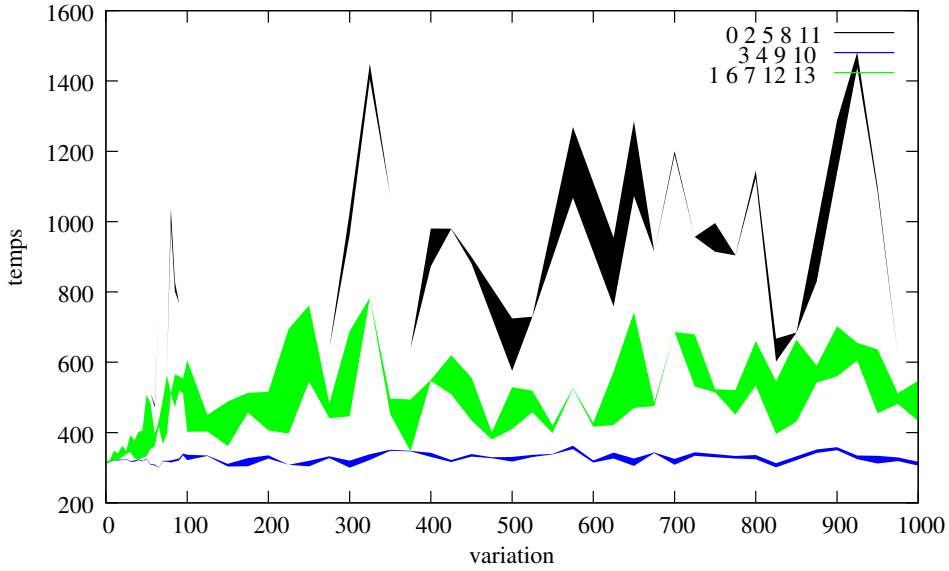


FIG. 5.10 – Temps d'exécution de 40000 requêtes en fonction de la variation entre les fréquences d'apparition de l'ensemble de requêtes initial et celles de l'ensemble des requêtes envoyées à l'ordonnanceur pour différentes heuristiques. Les heuristiques ayant un comportement très proche ont été regroupées en une seule courbe.

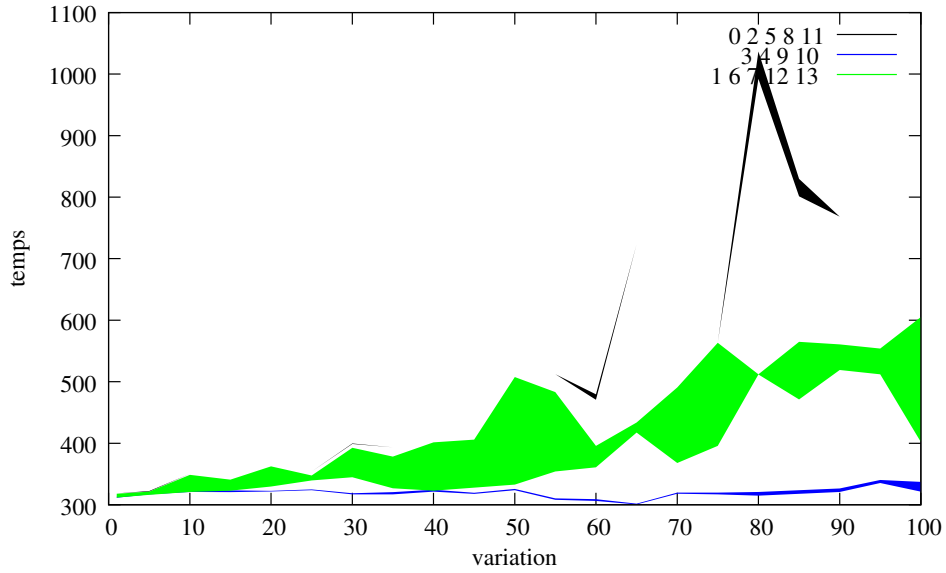


FIG. 5.11 – Zoom de la figure 5.10 sur l'intervalle [0..100]

peu de variations, toutes les heuristiques ont des résultats très proches mais dès que le facteur de variation augmente sensiblement, les différences se font très vite remarquer.

Dans le groupe ayant les plus mauvais résultats se trouve, comme nous pouvions

nous y attendre, l'heuristique 0 qui ne fait aucune adaptation du placement des données ou de l'ordonnancement. Les résultats de cette heuristique 0 montre, s'il en était besoin, qu'une adaptation dynamique est nécessaire dans le cas où les requêtes qui arrivent ne suivent pas les prévisions initiales. Dans le même groupe, nous trouvons toutes les combinaisons d'heuristiques utilisant **LPSched** pour le réordonnancement des tâches. Cela peut paraître surprenant car cette heuristiques devrait assurer, normalement, le meilleur ordonnancement possible pour le placement qui vient d'être effectué, nous pouvions donc nous attendre à des performances supérieures aux autres heuristiques. Trois phénomènes permettent d'expliquer ces mauvais résultats. Le premier est le coût du calcul du nouvel ordonnancement. En effet, la résolution du programme linéaire par ***lp_solve*** , la lecture et l'analyse de la solution prend une dizaine de secondes (sur la machine réalisant les simulations). Puisque il s'agit d'un réordonnancement global, pendant ce temps de calcul, aucune nouvelle requête ne peut être ordonnancée, ni aucune requête en queue sur les serveurs ne peut être envoyée vers des nœuds de calcul disponibles. Le deuxième point est lié au premier, il est dû au fait qu'il s'agisse d'un réordonnancement global, donc à priori l'ensemble des informations d'ordonnancement sont susceptibles d'être modifiées pour être en adéquation avec le nouveau placement des données. C'est-à-dire qu'une fois le nouvel ordonnancement calculé, il faut resoumettre à l'ordonnanceur l'ensemble des requêtes qui sont en attente dans les queues de traitement afin qu'elles puissent être envoyées aux serveurs de calcul les plus aptes à les traiter conformément au nouveau placement et aux nouvelles informations d'ordonnancement. Cela entraîne qu'à chaque phase d'adaptation, un grand nombre de requêtes devront retourner à l'ordonnanceur principal pour y être réordonnancées. Même si la phase d'ordonnancement est quasiment instantanée, il y a là encore une légère perte de temps de calcul. Enfin, le troisième point et le plus coûteux, provient de la nature même des informations utilisées pour la génération du programme linéaire et explique la si grande proximité de résultat avec l'heuristique 0. Après une phase de mouvement de données, le nouvel ordonnancement est calculé à partir des fréquences d'utilisation des requêtes réellement observées dans la plateforme. Or il faut qu'un grand nombre de requêtes ait déjà été soumise pour que les fréquences observées reflètent fidèlement les fréquences réelles du jeu de requête. Il y a alors toute une période où l'ordonnancement se fait avec des informations erronées et du même coup entraîne un ordonnancement de mauvaise qualité qui va avoir les mêmes effets que si l'ordonnancement n'avait jamais été modifié.

Le dernier groupe comprend les heuristiques 1, 6, 7, 12 et 13. Les heuristiques 6, 7, 12 et 13 ont comme point commun d'être basées sur **MaxDiff** et **LessUsed** pour le choix des données à dupliquer ou supprimer et de ne pas utiliser **LPSched** comme technique de réordonnancement.

De la façon dont sont composés les trois groupes, nous pouvons dire que lorsque aucune des requêtes ne suit le schéma d'utilisation utilisé pour le placement et l'ordonnancement initiaux, dans nos conditions d'utilisation, les points critiques sont essentiellement le choix des données et la façon de réordonnancer. Dans ce cas, toutes les heuristiques utilisant **LPSched** (2, 5, 8 11) obtiennent systématiquement de mauvaises

performances, alors que celle basée sur **MaxCompData** et **MinCompData** réussissent à conserver de bons résultats indépendemment de la façon dont sont choisis les serveurs à étudier ou de comment se fait le réordonnancement (à l'exclusion de ceux utilisant **LPSched** ).

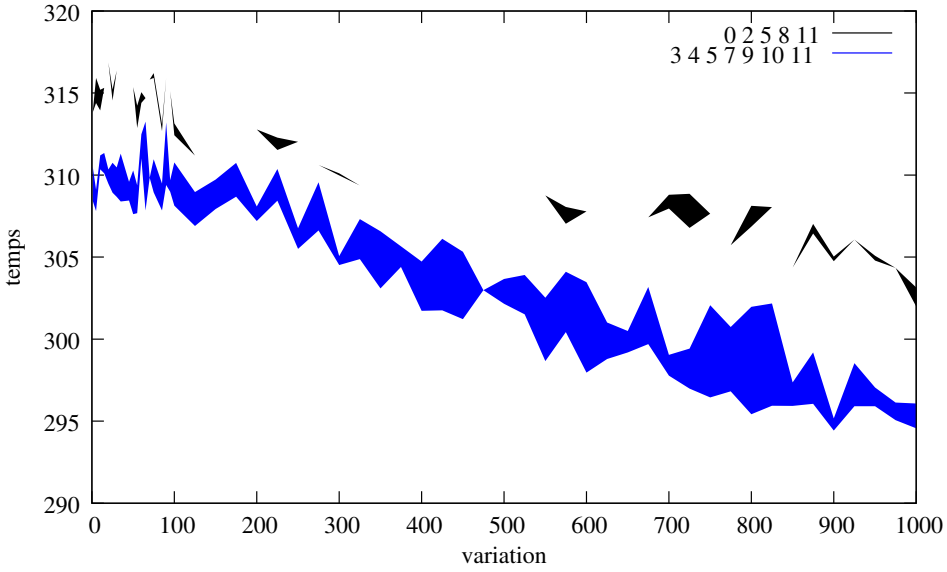


FIG. 5.12 – Temps d'exécution de 40000 requêtes en fonction de la variation entre les fréquences d'apparition de l'ensemble de requêtes initial et celles de l'ensemble des requêtes envoyées à l'ordonnanceur pour différentes heuristiques. Seuls les 10 types de requêtes les plus fréquentes ont eu leur fréquence d'utilisation augment. Les heuristiques ayant un comportement très proche ont été regroupées en une seule courbe.

Les figures 5.12 représentent les résultats obtenus lorsque seules les fréquences d'apparition des dix types de requêtes les plus présentes sont modifiées. Sur la figure 5.12 nous pouvons noter que cette fois, il n'y a que deux groupes et que les écarts entre les deux sont bien moins grands que lorsque l'ensemble des fréquences subit des modifications. Mais la première chose qui peut surprendre sur cette courbe c'est que plus le facteur de variation est grand, plus le temps d'exécution de l'ensemble du jeu de requête diminue. Cela s'explique par le fait des caractéristiques des requêtes les plus fréquentes. En effet, il s'agit essentiellement de requêtes nécessitant moins de temps de calcul que la moyenne des requêtes, soit parce qu'elles utilisent des banques de données de petites tailles, soit parce que les algorithmes concernés sont peu consommateurs en terme de ressources de calculs. Ainsi, en augmentant le nombre de ces requêtes dans le flot, le temps de calcul nécessaire pour traiter l'ensemble des requêtes est plus faible que celui nécessaire pour traiter le jeu de requêtes qui a permis d'établir les fréquences initiales.

Ce qui est à remarquer dans ces courbes, c'est que le groupe obtenant les moins bonnes performances est encore une fois constitué par l'heuristique 0 et celles utili-

sant **LPSched** pour le réordonnancement. Toutes les autres obtiennent des résultats très proches. Ainsi, le facteur déterminant dans ce cas est la méthode utilisée pour le réordonnancement des requêtes. Dans cette série d'expériences, plus le facteur de variation est important, plus les requêtes concernent presque exclusivement un très petit nombre de données de taille légèrement inférieure à la moyenne qui étaient, dans le placement initial, déjà largement répliquées (car largement utilisées). Ainsi, l'essentiel de l'adaptation va être de faire face à la saturation des serveurs causée par le nombres excessifs de requêtes qui arrivent. Là encore, **LPSched** pèche par son manque de connaissance de la réalité, alors que les méthodes simples d'équilibrage de charge entre les serveurs surchargés et sous-chargés permettent d'équilibrer les charges des différents nœuds.

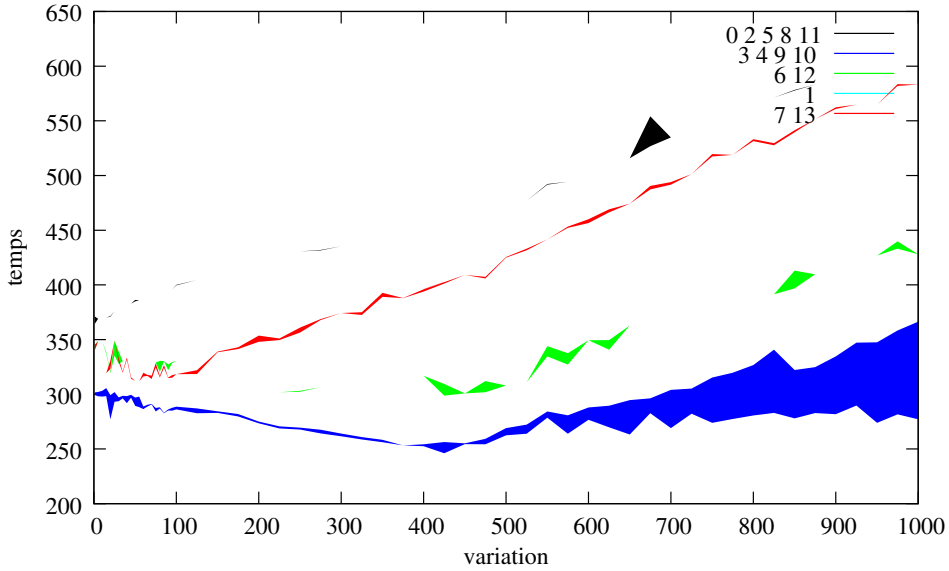


FIG. 5.13 – Temps d'exécution de 40000 requêtes en fonction de la variation entre les fréquences d'apparition de l'ensemble de requêtes initial et celles de l'ensemble des requêtes envoyées à l'ordonnanceur pour différentes heuristiques. Seuls les 10 types de requêtes les moins fréquentes ont eu leur fréquence d'utilisation augmentée. Les heuristiques ayant un comportement très proche ont été regroupées en une seule courbe.

Les figures 5.14 et 5.13 représentent les résultats obtenus lorsque seules les fréquences d'apparition des dix types de requêtes les moins représentées dans l'échantillon initial sont modifiées. Ces conditions d'expérimentation montrent des comportements très différents selon les heuristiques utilisées. Cette fois, cinq groupes distincts se démarquent, mais les meilleurs performances et les moins bonnes sont toujours réalisés par les mêmes groupes d'heuristiques. La encore, l'heuristique 0 et celles employant **LPSched** obtiennent les plus mauvaises performances, les meilleures étant obtenues par les heuristiques 3, 4, 9 et 10 utilisant toutes **MaxCompData** et **MinCompData** et soit **EquShareSched** soit **MedianCompSched** pour le réordonnancement.. Dans cette série de simulation nous pouvons constater trois phases différentes.

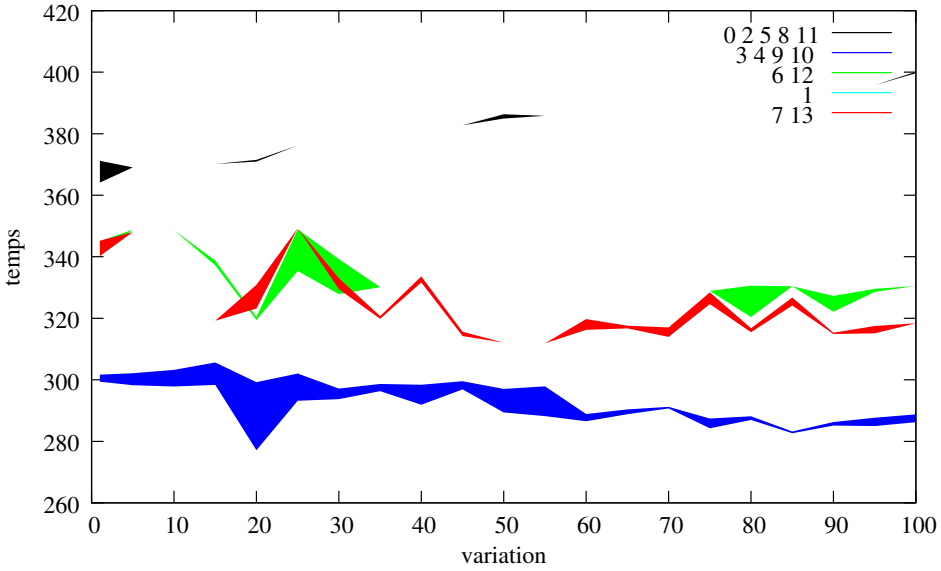


FIG. 5.14 – Zoom de la figure 5.13.

La première phase se situe pour les facteurs de variations situé entre $0$ et $100$. Comme nous le voyons sur la figure 5.14, zoom sur $[0..100]$ de la figure 5.13, pratiquement l'ensemble des heuristiques arrivent à maintenir leur temps d'exécution à l'identique, à l'exception, bien sur, des heuristiques $0$, $2$, $5$, $8$ et $11$ dont les performances commencent déjà à s'écrouler.

Ensuite arrive une deuxième phase correspondant à des facteurs de variations entre $100$ et $400$. Dans cette zone, les heuristiques $1$, $3$, $4$, $6$, $9$, $10$ et $12$ voient le temps d'exécution de l'ensemble des requêtes aller en diminuant. Comme dans le cas précédent, cela s'explique par les caractéristiques des requêtes dont la fréquence a augmenté. Il s'agit ici des requêtes les moins fréquemment utilisées dans le jeu de requêtes issu des traces d'exécution réelles. Pour la plupart ces requêtes concernent des banques de données occupant beaucoup plus d'espace de stockage que la moyenne et dont l'utilisation est vraiment très faible. Lorsqu'une de ces banques de données est placée sur un serveur, celui-ci lui devient presque dédié, n'ayant pas assez de place pour stocker d'autres banques. Or, l'utilisation de cette banque étant rare, même si elle est coûteuse, elle ne suffit pas à utiliser à plein temps le serveur qui l'héberge. En augmentant le nombre des requêtes concernant ces banques de données coûteuses en espace, nous obtenons une meilleure utilisation des serveurs qui les stockent à condition, bien sur, d'effectuer une adaptation efficace de l'ordonnancement. De plus, dans cette même zone, l'utilisation des autres banques devient fortement minoritaire, elles peuvent donc être supprimées ou déplacées assez facilement pour permettre le stockage des grosses banques qui sont de plus en plus demandées. De cette façon, la plate-forme devient presque dédiée à quelques grandes banques de données et est capable de les traiter efficacement.

Dans la dernière phase, le taux de requêtes sur les très grosses banques de données excèdent les capacités de traitement de la plate-forme et le temps d'exécution global augmente.

#### 5.5.4.3 En fonction de l'espace de stockage

Dans cette série de simulations, nous avons cherché à déterminer l'impact que pouvait avoir les caractéristiques de l'espace de stockage sur les heuristiques. Pour cela, nous avons fixé le facteur de variations à une valeur de $500\%$. Il s'agit là d'une valeur relativement élevée qui implique que le jeu de requête soumis n'a plus que peu de ressemblances avec le jeu de requêtes utilisé pour le placement initial. Mais avec une telle valeur nous savons, grâce aux simulation précédentes, que les différents groupes d'heuristiques ont déjà des comportements distincts et relativement stables. Ainsi, nous pourrons correctement voir l'effet de la taille de l'espace de stockage sur les différentes heuristiques. En ce qui concerne le réseau d'interconnexion des nœuds, là encore, les paramètres utilisés sont ceux fournis par le générateur de topologie Tiers.

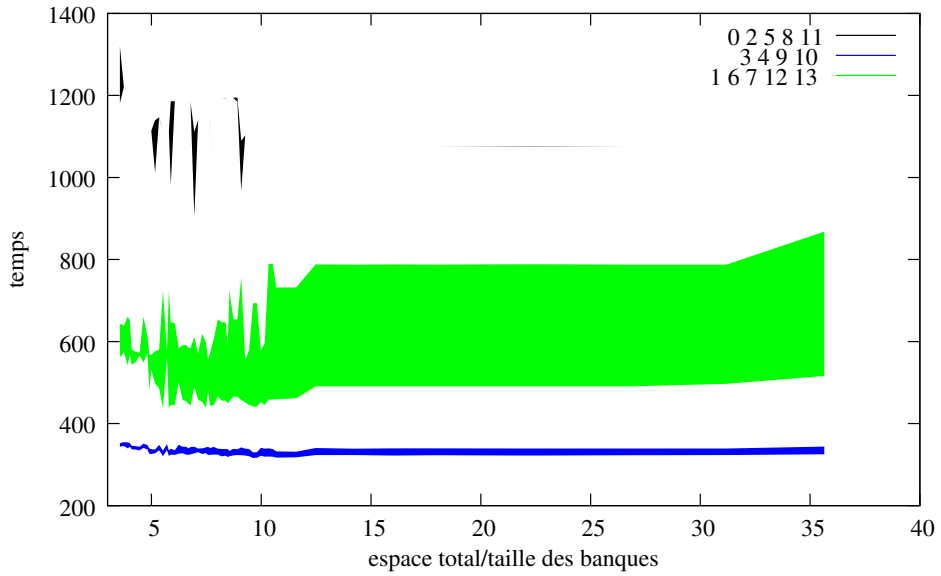


FIG. 5.15 – Temps d'exécution de $40000$ requêtes en fonction du rapport entre l'espace de stockage total disponible dans la plate-forme et le volume que représente l'ensemble des données à stocker pour les différentes heuristiques lorsque tous les types de requêtes sont affectés par le facteur de variation.

La figure 5.15 représente les résultats obtenus lorsque le facteur de variation est appliqué à tous les types de requêtes, les figures 5.17 (resp. 5.19) sont les résultats lorsque le facteur de variation est utilisé uniquement pour modifier les fréquences des dix types de requêtes les plus (resp. moins) présents dans l'échantillon initial. Les figures 5.16, 5.18 et 5.20 sont des zoom sur l'intervalle $[0..13]$ des figures 5.15, 5.17 et 5.19.

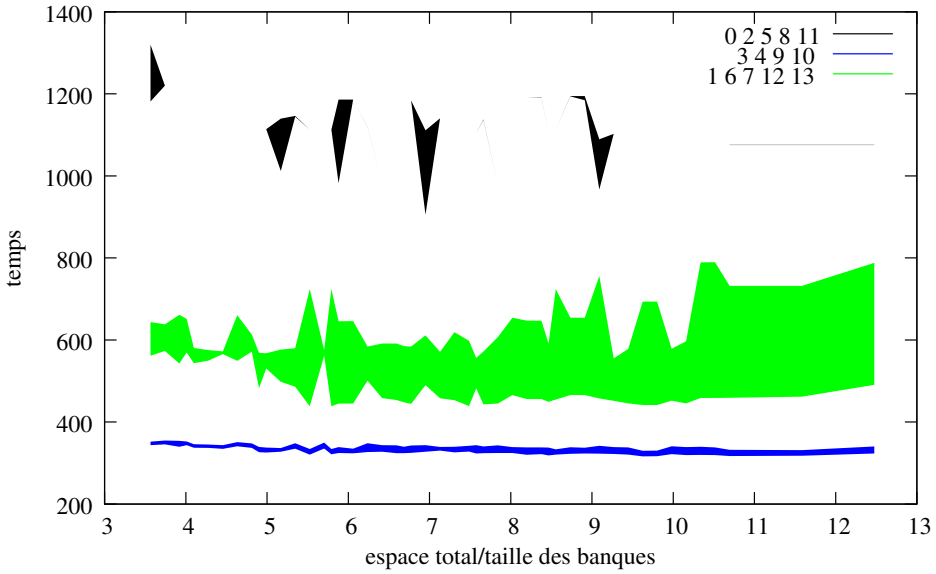


FIG. 5.16 – Zoom sur la zone [1..12] de la figure 5.15

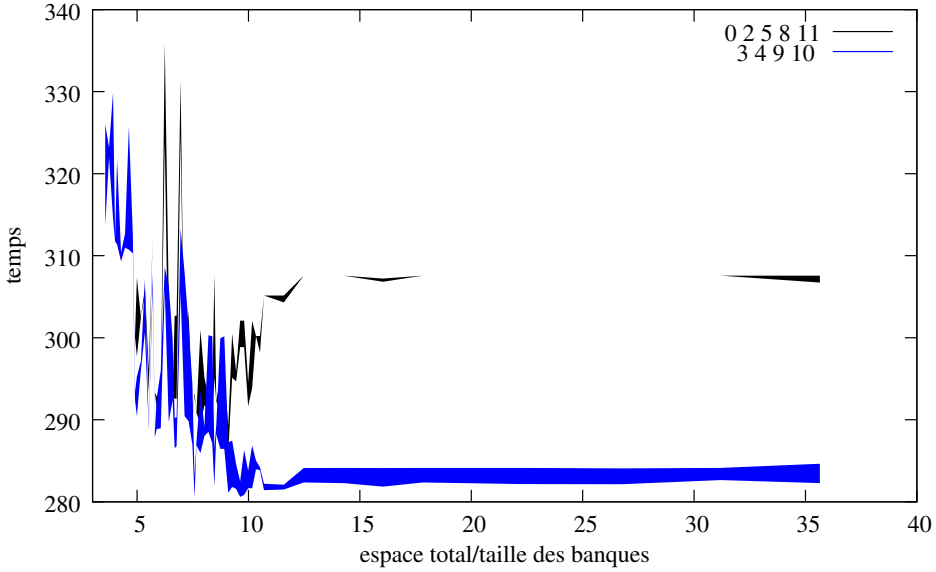


FIG. 5.17 – Temps d'exécution de 40000 requêtes en fonction du rapport entre l'espace de stockage total disponible dans la plate-forme et le volume que représente l'ensemble des données à stocker pour les différentes heuristiques lorsque seuls les 10 types de requêtes les plus fréquentes sont affectés par le facteur de variation.

Comme dans les expériences précédentes, l'ordonnée représente le temps d'exécution de l'ensemble du jeu de requêtes, par contre en abscisse se trouve le rapport entre l'espace de stockage total disponible sur la grille et le volume de données que re-

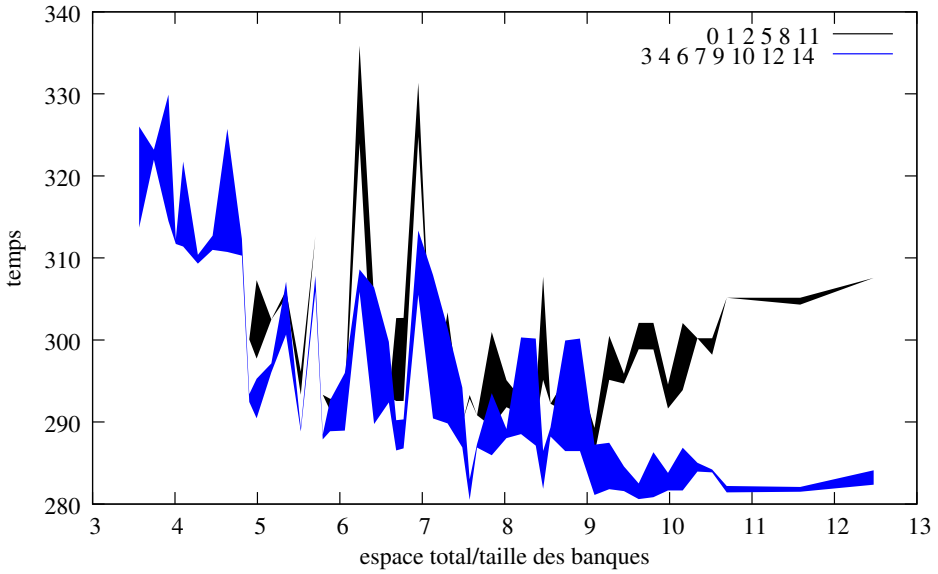


FIG. 5.18 – Zoom sur la zone [1..12] de la figure 5.17.

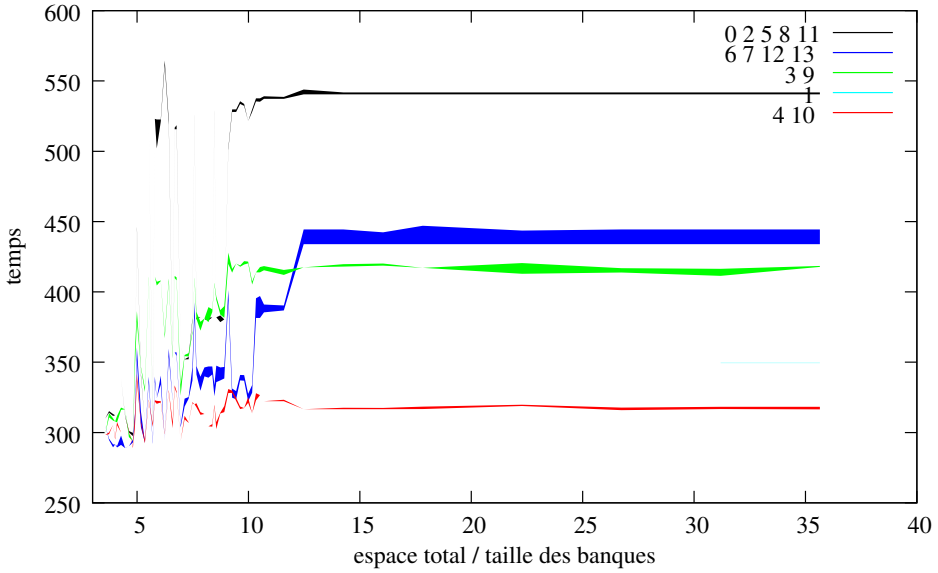


FIG. 5.19 – Temps d'exécution de 40000 requêtes en fonction du rapport entre l'espace de stockage total disponible dans la plate-forme et le volume que représente l'ensemble des données à stocker pour les différentes heuristiques lorsque seuls les 10 types de requêtes les moins fréquentes sont affectés par le facteur de variation.

présente les banques de données. Pour gagner en lisibilité, pour toutes ces figures, nous avons regroupé les courbes des heuristiques ayant des résultats similaires au sein d'une même courbe.

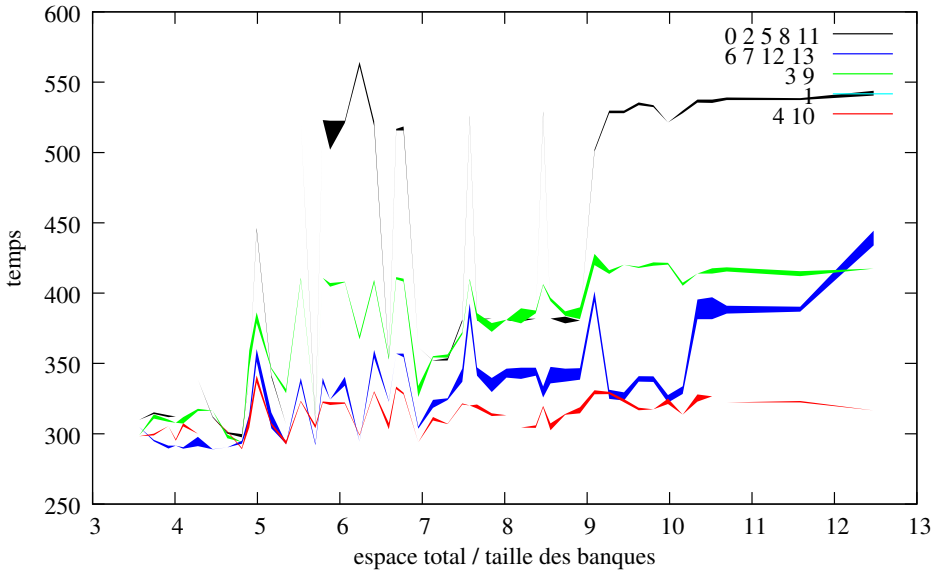


FIG. 5.20 – Zoom sur la zone [1..12] de la figure 5.19.

Globalement, dans les trois cas de figure, nous retrouvons les mêmes résultats que précédemment dès que l'espace de stockage disponible dépasse 13 fois le volume des données à stocker. Dans cette zone, les heuristiques 0, 2, 5, 8 et 11 obtiennent toujours les plus mauvais résultats, alors que les heuristiques 3, 4, 9 et 10 obtiennent presque toujours les meilleures performances, 3 et 9 seront toutefois dépassées par l'heuristique 1 (dont la seule différence avec 3 est de ne pas effectuer de suppression de données) lorsque le facteur de variation est appliqué aux requêtes les moins fréquentes. Associés avec les résultats de 4 et 9, nous pouvons en conclure que la décision de suppression et la technique de réordonnancement des requêtes sont les deux points clés dans cette situation.

Dans ce cas, où les requêtes originellement les moins utilisées se retrouve les plus fréquentes, il est surprenant de constater qu'avec un espace de stockage suffisamment large pour stocker la majorité des banques sur tous les sites, les résultats obtenus sont moins bons qu'avec une zone de stockage réduite. Cela s'explique par le même phénomène que dans les simulations précédentes, c'est-à-dire les caractéristiques des banques de données en cause. Les requêtes les moins fréquentes sont principalement celles qui utilisent les banques de données les plus grandes. Lorsqu'il y a autant de place que possible, ces grosses banques de données sont répliquées sur tous les sites en laissant les autres données en place. Or, ces transferts ont un coût très élevé qui, au final, n'est pas rentabilisé. Lorsqu'il y a peu de place, même une fois supprimée la majorité des réplicats des petites banques, les grandes banques ne peuvent pas être répliquées partout. Ainsi, le coût de transfert est moindre et surtout le placement des données se stabilise assez rapidement, l'essentiel de l'adaptation du système concerne alors l'ordonnancement des requêtes avec les nouvelles conditions. Cet ordonnance-

ment est alors régulièrement affiné pour aboutir à une solution efficace.

Par contre, lorsque ce sont les requêtes les plus utilisées qui voient leurs fréquences croître, le temps global d'exécution diminue avec l'augmentation de l'espace de stockage. Là encore, c'est la taille des banques de données concernées qui en est principalement responsable. Il s'agit de banques de données de petites tailles dont le coût de transfert n'est pas excessif et qui peuvent alors être répliquées sur tous les sites et permettre l'utilisation des nœuds de calcul au maximum de leur capacités.

Dans l'intervalle $[1..12]$, comme il est possible de le voir sur les figures 5.16, 5.18 et 5.20 les résultats sont moins stables. Si dans le cas où tous les types de requêtes sont modifiés, trois groupes bien distincts sont remarquables, il n'en est pas de même dans les deux autres catégories de simulations. Lorsque seule une partie des types de requêtes voient leur fréquences modifiées, une même heuristique peut obtenir des résultats vraiment très différents. Dans ces conditions de stockage restreint, le placement d'une donnée à un mauvais endroit peu avoir d'importantes répercussions sur les performances de l'ensemble du système. Ainsi dans cette zone, le résultat des différentes heuristiques est plus aléatoire.

#### 5.5.4.4 En fonction du réseau

Après avoir étudié l'impact sur les performances des heuristiques de redistributions de l'écart entre le schéma d'utilisation théoriques des banques et des données et la réalité puis de l'espace de stockage disponible sur la plate-forme, nous nous sommes intéressé au débit du réseau d'interconnexion. Contrairement à la version statique de l'algorithme qui ne faisait aucun transfert de données, au cours de la phase dynamique des mouvements peuvent avoir lieu. Le débit du réseau peut donc intervenir dans l'efficacité des heuristiques. Dans cette série de simulations, nous avons alors fixé la variation de l'utilisation des requêtes à $500\%$ et l'espace de stockage globale à $15$ fois la somme des tailles des banques de données. Le paramètre changeant entre deux simulations étant alors le débit du réseau entre les nœuds. Nous avons décidé, pour pouvoir juger de l'impact plus efficacement, de considérer un réseau homogène où tous les liens ont la même capacité.

La figure 5.21 représente les résultats obtenus lorsque seul le facteur de variation est appliqué à tous les types de requêtes.

La figure 5.22 représente les résultats obtenus lorsque le facteur de variation est utilisé uniquement pour modifier les fréquences des dix types de requêtes les plus présents dans l'échantillon initial.

La figure 5.23 représente les résultats obtenus lorsque le facteur de variation est utilisé uniquement pour modifier les fréquences des dix types de requêtes les moins représentés dans l'échantillon initial.

Là encore, ces trois figures 5.21, 5.22 et 5.23 confirment les bons résultats des heuristiques $3, 4, 9$ et $10$ dans tous les trois cas considérés. Il est à remarquer qu'à quelques exceptions près, le débit du réseau n'a quasiment aucune influence sur les résultats

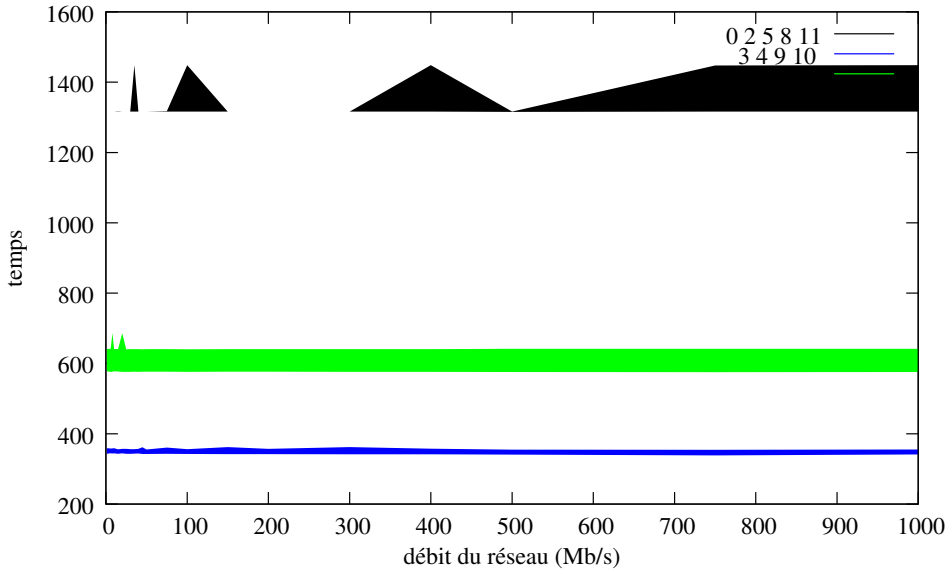


FIG. 5.21 – Temps d'exécution de 40000 requêtes en fonction du débit du réseau d'interconnection entre les nœuds de la plate-forme pour les différentes heuristiques lorsque tous les types de requêtes sont affectés par le facteur de variation.

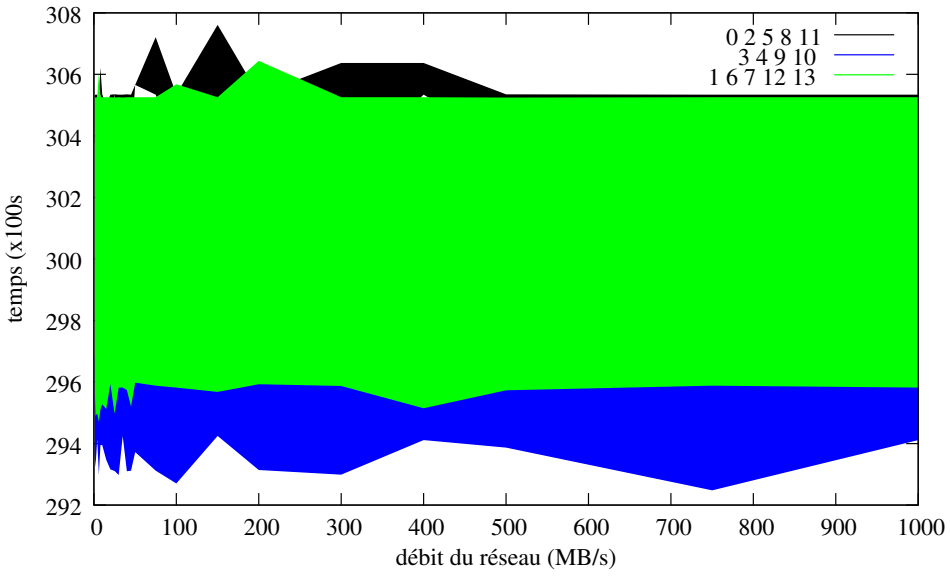


FIG. 5.22 – Temps d'exécution de 40000 requêtes en fonction du débit du réseau d'interconnection entre les nœuds de la plate-forme pour les différentes heuristiques lorsque seuls les 10 types de requêtes les plus fréquentes sont affectés par le facteur de variation.

des différentes heuristiques. En effet, les banques de données les plus fréquemment

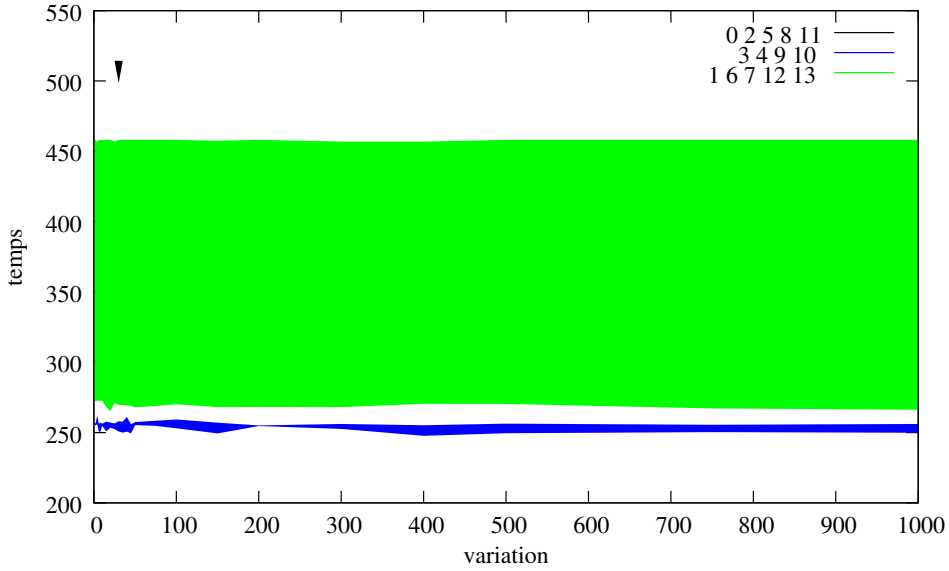


FIG. 5.23 – Temps d'exécution de 40000 requêtes en fonction du débit du réseau d'interconnection entre les nœuds de la plate-forme pour les différentes heuristiques lorsque seuls les 10 types de requêtes les moins fréquentes sont affectés par le facteur de variation.

utilisées sont rapidement placées sur une grande majorité des sites de calcul. Ainsi, le nombre de transferts reste relativement limité au cours du temps et le coût induit par ces transferts est très vite rentabilisé par les calculs. Nous constatons alors le même chose que dans les simulations précédentes à savoir que c'est essentiellement le choix des données et la façon dont sont réordonnancés les calculs qui font la différence entre les différentes heuristiques.

## 5.6 Conclusions

Dans ce chapitre nous avons commencé par présenter le simulateur de grille de données OptorSim et montré comment nous l'avons modifié afin de le rendre compatible avec les besoins de notre modèle d'exécution et nos algorithmes.

Avec ce simulateur, nous avons alors réalisé un premier ensemble de simulation pour valider l'algorithme SRA. Afin de mettre notre méthode en comparaison, nous avons établi un algorithme glouton pour déterminer un placement des données. Cela nous a permis de déterminer l'efficacité de notre méthode lorsque l'espace de stockage ou le débit du réseau d'interconnexion entre les nœuds ne sont pas trop élevés. C'est à dire que dans le cas où le transfert de données est coûteux et qu'il n'est pas possible de stocker l'ensemble des données sur tous les serveurs de calcul, alors notre méthode permet des gains importants par rapport à des ordonnancements MCT et un

placement glouton. Dans cette série, nous avons également montré l'efficacité de notre technique d'approximation de la solution du programme linéaire.

Toujours par simulation, nous avons ensuite testé une sélection des heuristiques établies pour répondre aux variations dans l'utilisation des applications et des banques de données bioinformatique. Suivant les différents paramètres que nous avons testé, il ressort que les heuristiques utilisant **MaxCompData** et **MinCompData** pour le choix des données et **MedianCompSched** pour le réordonnancement.

| | serveur surchargé | données de surcharge | serveur sous-chargé | données à supprimer | ordonnancement |
|---|---|---|---|---|---|
| 0 | aucun | aucun | aucun | aucun | aucun |
| 1 | MaxTimeServ | MaxCompData | MinTimeServ | aucun | EqShare |
| 2 | MaxTimeServ | MaxCompData | MinTimeServ | MinCompData | LPSched |
| 3 | MaxTimeServ | MaxCompData | MinTimeServ | MinCompData | EqShareSched |
| 4 | MaxTimeServ | MaxCompData | MinTimeServ | MinCompData | MedianCompSched |
| 5 | MaxTimeServ | MaxDiff | MinTimeServ | LessUsed | LPSched |
| 6 | MaxTimeServ | MaxDiff | MinTimeServ | LessUsed | EqShare |
| 7 | MaxTimeServ | MaxDiff | MinTimeServ | LessUsed | MedianCompSched |
| 8 | MaxOverMedianServ | MaxCompData | MinOverMedianServ | MinCompData | LPSched |
| 9 | MaxOverMedianServ | MaxCompData | MinOverMedianServ | MinCompData | EqShare |
| 10 | MaxOverMedianServ | MaxCompData | MinOverMedianServ | MinCompData | MedianCompSched |
| 11 | MaxOverMedianServ | MaxDiff | MinOverMedianServ | LessUsed | LPSched |
| 12 | MaxOverMedianServ | MaxDiff | MinOverMedianServ | LessUsed | EqShare |
| 13 | MaxOverMedianServ | MaxDiff | MinOverMedianServ | LessUsed | MedianCompSched |

TAB. 5.2 – Les différentes combinaisons entre les heuristiques qui ont été implémentées dans le simulateur.

# Chapitre 6

# Implémentation

## 6.1 Introduction

Grâce aux simulations, nous avons pu estimer que nos algorithmes donnent de bons résultats, mais cela ne reste que des simulations donc des résultats dépendant d'un modèle et de la qualité du simulateur lui-même. Afin de valider nos méthodes, nous avons décidé d'implanter une partie des algorithmes dans un intergiciel de grille existant. Notre choix s'est porté sur deux environnements logiciels qui fonctionnent ensembles pour offrir une suite d'éléments compatibles avec nos besoins. Ces environnements sont DIET (Distributed Interactive Engineering Toolbox) et DTM (Data Tree Manager). DIET offre un ensemble d'outils logiciels permettant la construction, le déploiement et l'utilisation de serveurs de calculs. DTM, quant à lui, est un gestionnaire de données prévu pour fonctionner avec DIET.

## 6.2 DIET : Distributed Interactive Engineering Toolbox

### 6.2.1 Architecture

DIET [9] est construit autour d'un composant logiciel nommé *Server Daemons (SeD)*. L'ordonnanceur est réparti sur un ensemble hiérarchisée d'*Agents*. La figure 6.1 montre l'organisation hiérarchique de DIET. Un **client** est une application qui utilise DIET pour résoudre des problèmes. Différents types de clients sont en mesure de se connecter à DIET depuis une page web, un environnement de résolution de problème tel que Matlab ou Scilab, ou directement depuis un programme écrit en C ou en Fortran. Les calculs sont effectués par des serveurs que l'on atteint au travers d'un SeD. Par exemple, le SeD peut être situé sur la machine qui sert de point d'entrée à un calculateur parallèle. Les informations stockés par le SeD sont une liste des problèmes qu'il est possible de résoudre sur ce serveur et éventuellement les informations concernant sa charge actuelle (mémoire libre, nombre de ressources disponibles, ...). Un

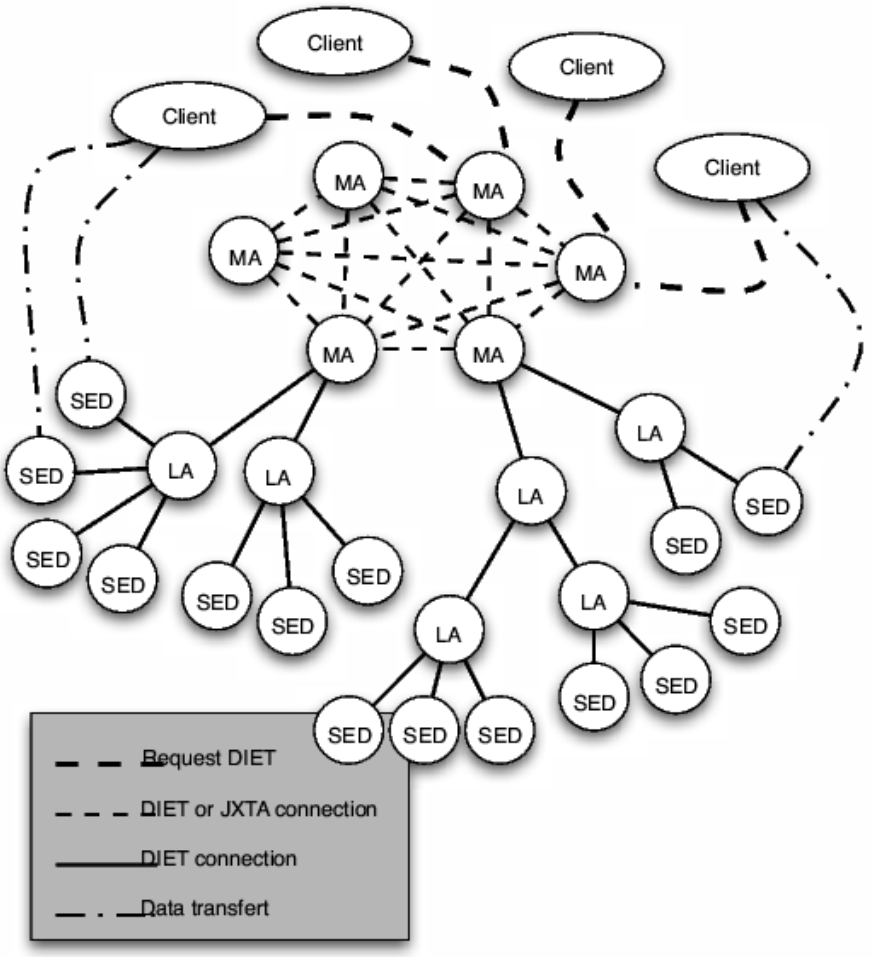


FIG. 6.1 – L'organisation hiérarchique de DIET.

SeD déclare les problèmes qu'il peut résoudre à son parent dans la hiérarchie : le *Local Agent(LA)*. La hiérarchie des agents d'ordonnancement est composée d'un ou plusieurs **Master Agent(MA)** et éventuellement d'un ou plusieurs **Local Agent (LA)**.

Un MA est un point d'entrée de l'environnement. Il reçoit les requêtes de calcul des clients. Ces requêtes font références à des problèmes DIET. Le MA collecte alors les capacités de calculs des serveurs et choisi le meilleur en accord avec les stratégies d'ordonnancement. Une référence vers le serveur choisi est alors retourné au client. Les agents ont pour rôle de transmettre les requêtes et les informations entre les MAs et les SeD. Les informations stockées au niveau des agents sont la liste des requêtes et le nombre de serveurs qui peuvent résoudre un problème donné. En fonction de la topologie du réseau, une hiérarchie d'agents peut être déployée entre le MA et les SeD.

DIET inclue un module appelé FAST (*Fast Agent's System Timer*) [16] qui fournit dif-

férentes informations requises par les agents. FAST permet à des applications clientes d'obtenir une prévision fiable des besoins de routines connues en terme de temps d'exécution, d'espace mémoire nécessaire ainsi que la disponibilité des machines (mémoire, charge cpu), et des coûts de communications. La prévision des temps de communications entre différents éléments de la hiérarchie est réalisée par l'utilisation de NWS (Network Weather Service) [37]. Des senseurs NWS sont placés sur chaque nœud de la hiérarchie pour récupérer des informations concernant la disponibilité des ressources, qui seront ensuite collectées et utilisées par FAST.

### 6.2.2 Créer une application grille avec DIET

L'idée principale est de fournir un certain niveau d'intégration pour des applications destinées à être exécutées sur une grille de calcul. La figure 6.2 montre les différents niveaux. L'interface de plus haut niveau fournit un accès à la grille directement depuis une page web ou un langage de haut niveau tel que Scilab. Lorsqu'une interface est disponible, elle est construite sur la **couche client**.

La couche client est le lien entre l'interface de haut niveau et le client DIET. Elle doit être conçue par un utilisateur désireux d'offrir un nouveau service de grille. Une API est fournie pour pouvoir facilement développer une application cliente.

Le rôle majeur du client DIET est de soumettre des requêtes au Master Agent. En retour, il reçoit une liste de références de serveurs en mesure de résoudre son problème, cette liste est triée selon la politique d'ordonnancement de sorte que le premier SeD de cette liste soit celui qui est le plus apte, selon l'ordonnanceur, à traiter le problème efficacement. Finalement, le client DIET enverra les données nécessaires au calcul à ce serveur.

La hiérarchie d'agents doit trouver un serveur, ou une liste de serveurs, en mesure de traiter une requête spécifique (type de problème, capacités des ressources, statuts des serveurs, ...) et de retourner l'information au client DIET.

L'utilisateur qui a développé l'application cliente doit également écrire la partie correspondante à l'exécution du problème sur le serveur. Là encore, une API est fourni pour réaliser simplement une connexion entre l'application et le serveur DIET.

### 6.2.3 Exécution d'une requête dans DIET

La figure 6.3 montre les différentes étapes qui se déroulent lors de la soumission d'une requête par un client au MA. La première étape (1) est la soumission du problème ; pour cela le client se connecte au MA et lui transmet une description des différentes informations relatives au problème à résoudre. Le MA déclenche alors un parcours en profondeur de la hiérarchie DIET en s'adressant à ses fils en leur transmettant les informations nécessaires dans le but de trouver les SeD capables de résoudre le problème (étapes 2 et 3). Lorsque la requête arrive sur un SeD, celui-ci transmet à

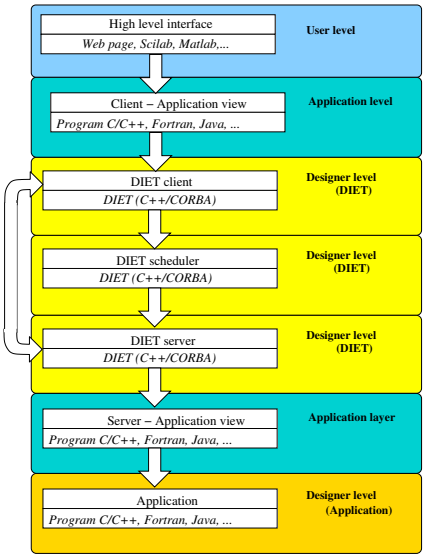


FIG. 6.2 – Les différentes couches d'interaction du noyau DIET jusqu'à l'application cliente.

son père sa capacité, ou non, à la résoudre ainsi que des informations de performances (étape 4). Lorsqu'un LA a reçu toutes les réponses de ses fils, il ordonne ses réponses en fonction de divers critères (utilisation des SeD, paramètres du problème, ...) (étape 5), et la transmet à son père (étape 6). Lorsque le MA a reçu les réponses de l'ensemble de ses fils, il ordonne à son tour les SeD capable de résoudre le problème (étape 7) et envoie au client une liste des serveurs les plus aptes à répondre à ses attentes (étape 8). A l'aide de cette liste, le client contacte directement le serveur et lui soumet sa requête et transfert ses données si nécessaire (étape 9). Une fois que le SeD choisi a récupéré tout ce dont il a besoin pour exécuter la requête, il traite celle-ci (étape 10) et envoie les résultats directement au client (étape 11).

## 6.3 DTM : Data Tree Manager

Le service de gestion de données DTM a été développé spécifiquement pour la plate-forme DIET. Ce système propose une gestion des données basée sur deux éléments clés : des identifiants pour les données et une gestion en arbre. Afin d'éviter de multiples transmissions de la même donnée d'un client vers les serveurs de calcul, DTM offre la possibilité de laisser des données à disposition à l'intérieur de la plate-forme après la fin des calculs. Dans ce cas, un identifiant est attribué à la donnée et retourné au client qui pourra, par la suite s'en servir pour faire référence à sa donnée lors d'une utilisation ultérieure. Un client peut choisir, pour chacune de ses données, si celle-ci doit être persistante ou non à l'intérieure de la plate-forme. Plusieurs modes de persistance sont proposés et sont décrits dans la table 6.1.

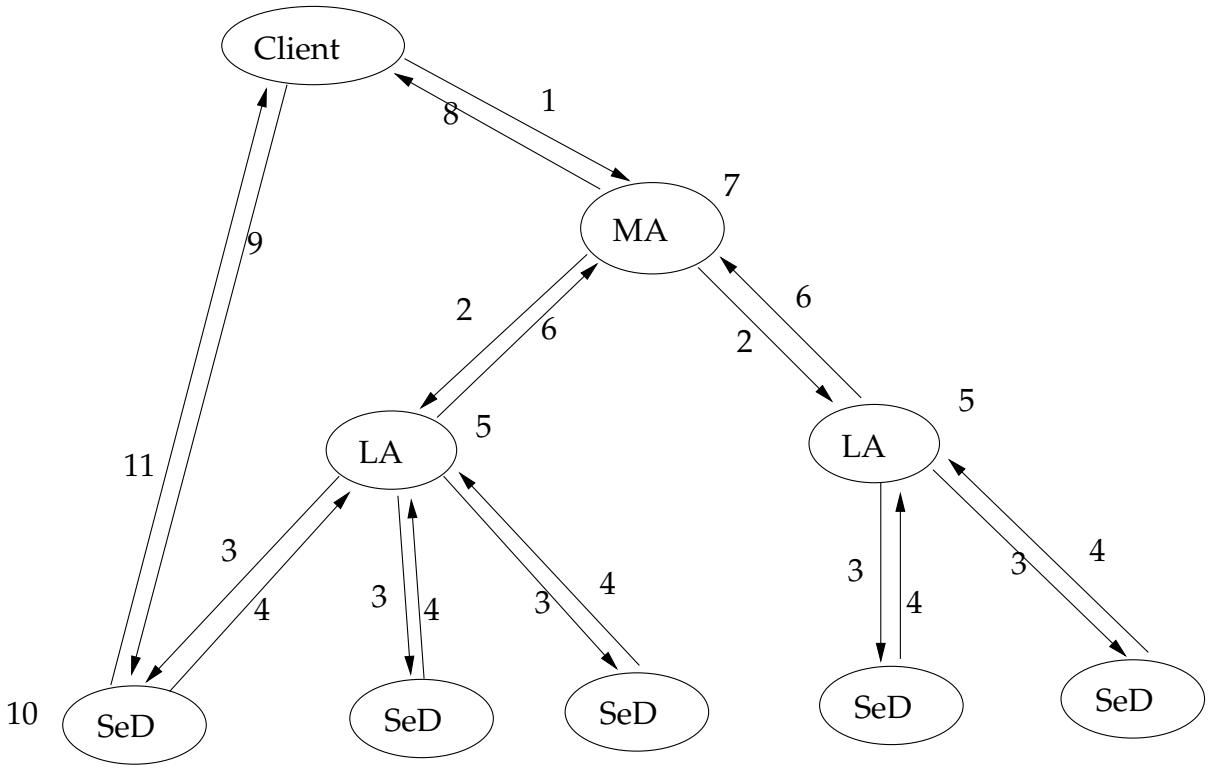


FIG. 6.3 – Les différentes étapes lors de l'exécution d'une requête dans l'architecture DIET.

| mode | description |
|---|---|
| DIET_VOLATILE | non persistant |
| DIET_PERSISTENT | persistant et déplaçable |
| DIET_PERSISTENT_RETURN | persistant, déplaçable et une copie est envoyée au client |
| DIET_STYCKY | persistant mais ne peut être déplacer sur un autre serveur |
| DIET_STICKY_RETURN | persistant, ne peut être déplacer, une copie est envoyée au client |

TAB. 6.1 – Les différents modes de persistance des données.

Pour éviter un entrelacement des messages concernant les données et ceux concernant les requêtes de calcul, l'architecture proposée sépare totalement la gestion des données de celle des requêtes. DTM a une structure hiérarchique en arbre plaquée sur celle de DIET. Comme nous peuvons le voir dans la figure 6.4, il est construit autour de trois entités, le *Logical Data Manager*, le *Physical Data Manager* et le *Data Mover*.

**Logical Data Manager** Le *Logical Data Manager* est composé d'un ensemble d'objets *LocManager*. Un LocManager est situé sur chaque agent DIET avec lequel il communique directement. Ils sont donc organisés hiérarchiquement en arbre comme les agents DIET. La figure 6.5 montre le lien entre les SeD et la structure hiérarchique.

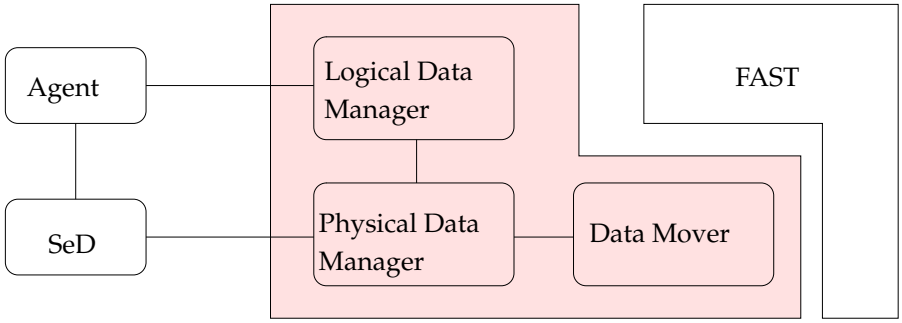


FIG. 6.4 – L'architecture du Data Tree Manager.

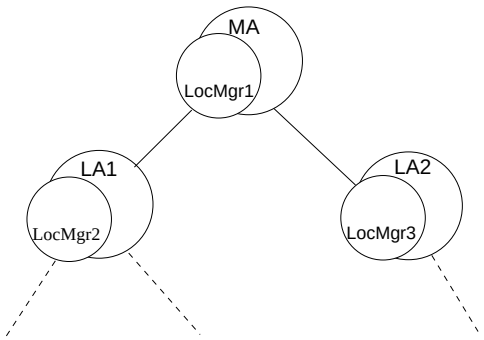


FIG. 6.5 – La connexion entre les Agents DIET et les LocManager.

Les LocManager ont la charge de la localisation des données dans la hiérarchie. Ainsi, chaque LocManager stocke une liste des données qui sont présentes dans le sous-arbre qui se trouve en dessous de lui dans la hiérarchie.

**Physical Data Manager** La deuxième entité est le *Physical Data Manager* qui est constitué d'un ensemble d'objets *DataManager*. Les DataManager sont situés sur chaque SeD avec lequel il communique localement. Il stocke les données et maintient une liste de celle-ci. Lorsque le SeD auquel il est associé a besoin d'une donnée, il la lui transmet. Enfin, il informe le LocManager situé juste au dessus de lui dans la hiérarchie des opérations d'ajouts et de suppressions de données. La figure 6.6 montre le lien entre les SeD DIET et les DataManager.

**DataMover** En dernier lieu, le *Data Mover* permet le déplacement des données entre les DataManager. C'est l'élément qui fournit les mécanismes de transfert de données entre les DataManager. Il a aussi en charge de lancer la mise-à-jour des DataManager et des LocManager lorsque le transfert d'une donnée est terminé.

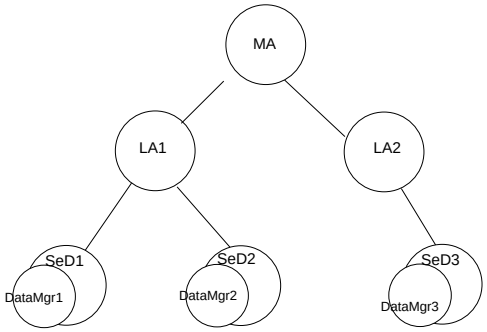


FIG. 6.6 – La connexion entre les SeD DIET et les DataManager.

## 6.4 Implémentation des algorithmes

### 6.4.1 Introduction

Comme nous l'avons vu au cours de ce chapitre, l'utilisation conjointe de DIET et DTM nous permet d'avoir pratiquement tout l'environnement nécessaire à la mise en œuvre des algorithmes que nous avons présentés dans le chapitre 4. Cependant, il est tout de même nécessaire d'effectuer des modifications au cœur du Master Agent de DIET afin de pouvoir mettre en œuvre les mécanismes d'ordonnancement. En effet, si notre méthode est conçue pour venir se placer au dessus de l'ordonnanceur de la grille et du gestionnaire de données, l'ordonnanceur de DIET n'est pas prévu pour faire appel à un système d'information extérieur. Nous avons donc greffé directement notre mécanisme d'ordonnancement à l'intérieur du Master Agent, nous verrons ultérieurement les détails. Pour l'interfaçage avec le DTM, les choses n'ont posé aucun problème puisque, celui-ci possède l'API dont nous avons besoin pour préciser le placement des données et donner les ordres de déplacements des données.

### 6.4.2 L'implémentation

Puisque le DTM possède déjà une interface permettant, depuis le MA de donner des ordres de transfert de données, nous n'avions rien de plus à modifier pour cette phase. La partie la plus importante a donc été d'adapter le MA et les mécanismes de DIET afin de pouvoir intégrer notre technique d'ordonnancement. Les méthodes liées à l'expérimentation de SRA peuvent se décomposer en trois parties.

Tout d'abord la phase d'initialisation du système, qui consiste, à partir des informations connues sur les fréquences des requêtes et la plate-forme d'exécution, à calculer un premier placement ainsi que des informations d'ordonnancement. La deuxième partie est la phase qui concerne la soumission et l'exécution des requêtes. Lorsqu'un client soumet une requête au MA, celui-ci doit trouver et lui retourner la référence du

SeD qui se chargera d'exécuter la requête. En dernier lieu se trouve l'exécution de la requête.

#### 6.4.2.1 Initialisation

L'un des éléments de cette étape est de pouvoir récupérer les références des SeD qui ont été déployés et qui sont pris en compte par l'algorithme d'ordonnancement et de placement. En effet, pour pouvoir ordonner des transferts ou permettre la soumission des requêtes par les clients, il est indispensable d'obtenir ces identifiants. Pour cela, une fois que la plate-forme DIET est lancée et avant le traitement des requêtes des clients, une requête concernant un problème défini sur tous les SeD sera envoyée et traitée pratiquement comme une requête DIET normale. La soumission de cette première requête aura pour conséquence de provoquer un parcours de la hiérarchie DIET et de retourner au MA les références de l'ensemble des SeD permettant de pouvoir s'y connecter.

Les informations concernant la plate-forme et l'utilisation des données sont connues à l'avance. Grâce à elle, il est possible de générer le placement et l'ordonnancement.

#### 6.4.2.2 Le cycle d'une requête

La figure 6.7 montre le cycle de vie d'une requête dans l'architecture que nous avons développée à partir de la hiérarchie DIET. Lorsqu'une requête est soumise par un client au MA (étape 1), celui-ci, dans le fonctionnement de DIET normal, lance un parcours de sa hiérarchie afin de trouver les SeD qui seront en mesure d'exécuter cette requête. Dans notre cas, ce parcours de l'arbre des agents n'est pas nécessaire puisque l'ordonnanceur connaît déjà toutes les informations nécessaires et peut donc renvoyer la référence du SeD directement. Ainsi, au lieu de lancer la recherche dans sa hiérarchie, le MA s'adresse directement à notre système (étape 2) qui lui retourne l'adresse du serveur sur lequel la requête devra être effectuée. Cette adresse est alors transmise au client (étape 3) et la suite de l'exécution suit son cours normal dans l'environnement DIET. C'est-à-dire qu'une fois que le client sait à quel serveur il doit s'adresser, il va contacter celui-ci et lui transmettre les informations nécessaires à la résolution de son problème (étape 4). Alors, le SeD concerné pourra effectuer le calcul requis (étape 5) et retourner au client les résultats du traitement (étape 6).

La seul différence par rapport au code original de DIET est le chemin que parcours la requête, ou plutôt celui qu'elle ne parcoure pas. Nos modifications dans le MA ont donc porté sur la dérivation de l'ordonnanceur afin que les requêtes n'aient pas à descendre dans la hiérarchie DIET.

**Remarque :** Si nous avons largement modifié le comportement normal de DIET afin qu'il soit en accord avec notre système d'ordonnancement, nous n'avons cependant

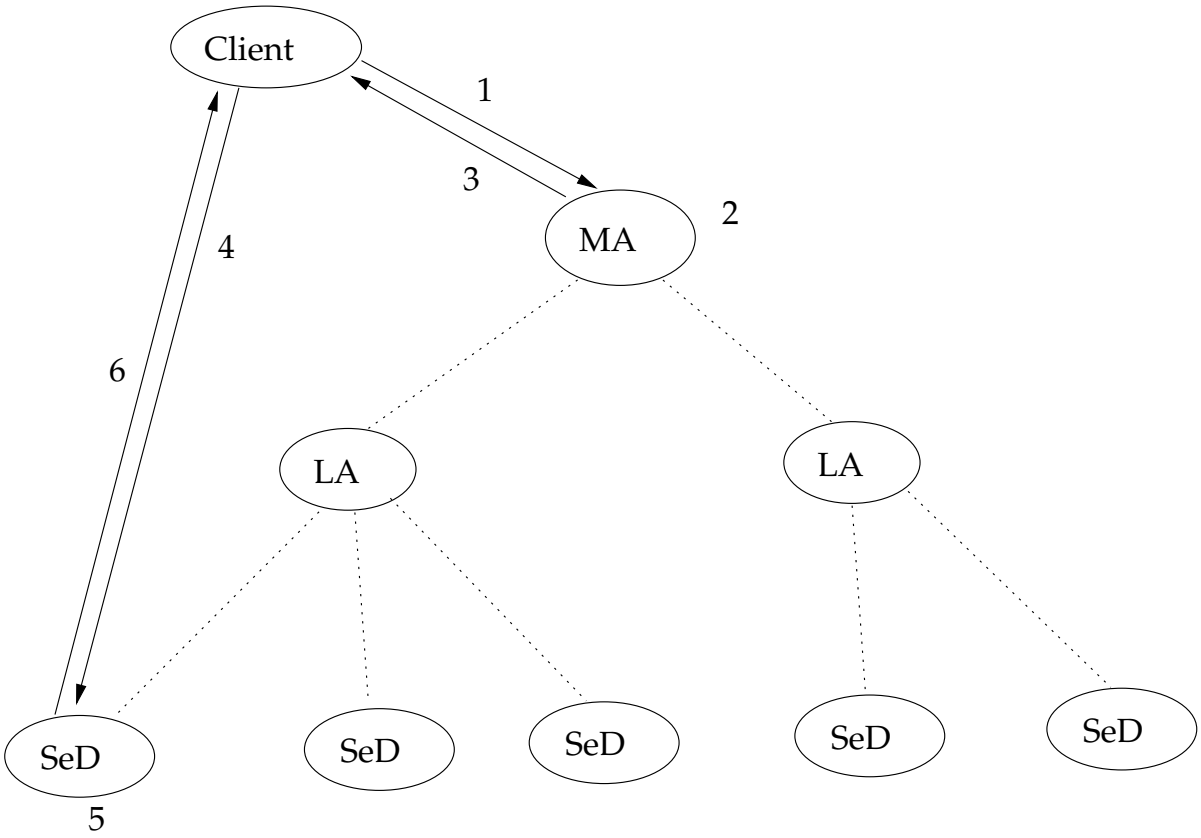


FIG. 6.7 – Le cycle d'une requête dans notre implémentation de DynSRA dans la version modifiée de DIET.

pas altéré ses fonctionnalités existantes. En fait, l'ordonnanceur que nous avons implanté au sein du MA n'est utilisé que si le nom du problème appelé est préfixé par «**bio-**». Si ce n'est pas le cas, alors c'est le fonctionnement normal de DIET qui est utilisé. Cette option sera très utile afin de pouvoir comparer notre ordonnanceur avec l'ordonnancement natif de DIET. C'est également une preuve, même si cela est dans un cas bien particulier, que notre système peut être ajouté à un système existant sans être trop intrusif et empêcher son fonctionnement normal.

**Remarque** L'exécution des algorithmes est simulée sur les SeD en suivant la modélisation qui nous a permis d'établir l'algorithme SRA, c'est-à-dire avec un coût en temps affine en la taille des données utilisées.

## 6.5 Expérimentations

### 6.5.1 Environnement

#### 6.5.1.1 La plate-forme *Grid'5000*

Les tests ont été effectués sur la grille expérimentale *Grid'5000* [26]. Le projet *Grid'5000* doit son nom à son objectif de vouloir déployer une grille de calcul pérenne comprenant 5000 processeurs répartis sur une dizaine de sites en France situés dans des laboratoires de recherche. Cette grille dédiée à la recherche dans tous les domaines liés aux grilles de calcul rend ce projet unique en son genre en faisant de *Grid'5000* un outil d'analyse et d'observation pour la communauté informatique dans le même esprit qu'un super télescope pour l'Astronomie ou le projet du grand collisionneur de Hadrons (*Large Hadron Collider - LHC*) pour le domaine de la Physique Nucléaire.

Ce projet est financé par le Ministère de la Recherche et de l'Education, l'INRIA, le CNRS, les Universités et les régions dont dépendent les institutions qui hébergent les nœuds de calcul. Actuellement, *Grid'5000* implique 17 laboratoires et comprend 2570 processeurs soit 1229 nœuds répartis sur 9 sites localisés à Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse. Les différents sites sont connectés par le réseau académique RENATER [36]. La figure 6.8 montre une carte de France schématique représentant les différents sites hébergeant les nœuds *Grid'5000* ainsi que les liens réseaux qui relient ces neufs sites.

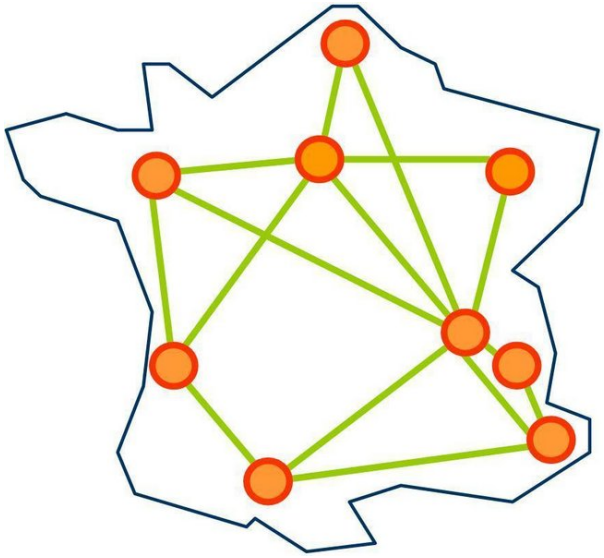

FIG. 6.8 – Représentation schématique des différents sites de *Grid'5000*.

L'outil *Grid'5000* est un environnement totalement dédié à la recherche sur les grilles et sécurisé, tout en offrant une souplesse d'utilisation. En effet, la plate-forme

*Grid'5000* n'est accessible que depuis des points d'entrée restreints situés sur les différents sites. Mais une fois connecté le réseau reliant les nœuds est totalement ouvert sans problème de système de pare-feu à l'entrée des différents sites. Par contre, l'utilisation des nœuds n'est possible qu'à la condition d'avoir effectué une réservation de ceux-ci par l'intermédiaire du gestionnaire de ressource OAR [7] disponible sur chaque site. Cette contrainte permet de garantir d'utiliser des nœuds qui seront dédiés le temps de l'expérience sans avoir le risque de « collision » avec l'expérience d'un autre utilisateur.

#### 6.5.1.2 Déploiement

Pour déployer une hiérarchie DIET il existe un logiciel GoDIET [8] qui, à partir de la description du déploiement souhaité au format XML se charge de transférer les différents exécutables nécessaires, de placer les variables d'environnement et de lancer la plate-forme. Cependant, puisque notre version de DIET utilise certains fichiers et informations qui ne font pas partie du système originel, il était plus simple de ne pas utiliser l'outil existant qu'il aurait fallu modifier en profondeur et développer des scripts spécifiques à notre système. Le déploiement de la plate-forme se fait alors en plusieurs étapes.

La première, et la plus simple, est la réservation des ressources de la grille. En effet, l'utilisation des nœuds de *Grid'5000* est contrainte par l'utilisation de OAR [7]. Il est à noter que bien qu'encore en cours de développement, *Grid'5000* est déjà largement utilisé et il peut parfois s'écouler beaucoup de temps avant de pouvoir avoir des nœuds sur l'ensembles des sites.

Une fois les nœuds attribués sur les différentes grappes qui constituent la grille, il faut récupérer l'espace de stockage disponible sur chacun d'entre eux. Nous considérons les nœuds qui hébergeront les SeD comme des éléments indépendant même si plusieurs d'entre eux se trouvent sur le même site. Cela signifie que les fichiers nécessaires seront stockés localement sur chacun des nœuds et non sur le serveur NFS commun aux nœuds d'un même site. Après avoir récupéré ces informations sur l'espace disponible, nous choisissons le nœud disposant du plus d'espace. Ce nœud sera celui qui hébergera les banques de données à l'initialisation de la plate-forme. D'ailleurs, il ne traitera aucune requête de calcul et ne sera même pas pris en compte lors de la résolution du programme linéaire calculant le placement et les informations d'ordonnancement.

La hiérarchie que nous allons déployer comporte un MA et des SeD. En effet, vu le nombre de nœuds utilisés, un peu plus d'une dizaine seulement, et la nature de notre algorithme, l'utilisation d'un ou plusieurs LAs aurait été complètement superflue. Parmi les nœuds dont nous disposons sur la grille, nous en choisissons alors un qui aura le rôle du MA. Tous les autres autres seront alors des SeD.

À partir des informations sur l'espace de stockage et les nœuds et celles concernant les requêtes et les banques de données, nous établissons le programme linéaire

associée à cette plate-forme et le résolvons en utilisant la méthode d'approximation décrite dans la section 4.3.4. Une fois le programme résolu, nous extrayons de la solution le placement et surtout les informations d'ordonnancement qui seront fournis à l'ordonnanceur rajouté dans la MA. La plate-forme peut alors être déployée sur les sites.

Une fois que la hiérarchie DIET est lancée, un premier client à la charge d'insérer dans la grille l'ensemble des données qui seront utilisées par la suite. Ces données sont initialement stockées sur le serveur qui a été dédié à cet usage.

#### 6.5.1.3 Visualisation des résultats : LogService et vizDiet

*LogService* [28] est un système de surveillance qui relaye les messages et les informations qui surviennent dans une plate-forme distribuée. Il s'agit d'un système relativement générique qui doit être interfacé avec l'application à surveiller. Pour cela chaque élément à monitorer se voit attacher un composant, nommé *LogComponent*, qui relate les évènements à un composant centralisé : le *LogCentral*. Celui-ci stocke l'information ou la transmet à des outils qui auront en charge de la traiter : les *LogTools*.

Le système LogService est mis en œuvre dans DIET [5] afin de pouvoir surveiller la plate-forme. La figure 6.9 montre l'architecture du système de monitoring intégré à DIET. Des LogComponents ont été implanté dans les Agents et dans les SeD et envoient leurs informations à un serveur LogCentral. Deux outils de type LogTools sont fournis avec DIET. Il s'agit de *DietLogTools* qui récupère et stocke l'ensemble des informations reçues par le LogCentral et de *vizDiet* dont le but principal est de fournir une représentation graphique et de suivre l'activité d'une plate-forme DIET. La figure 6.10 montre un ensemble de capture d'écran de vizDiet. VizDiet permet d'afficher beaucoup d'informations sur l'activité d'une plate-forme DIET comme la charge des nœuds au cours du temps, les flux de tâches ou encore des graphes de Gantt sur l'utilisation des serveurs. VizDiet peut se connecter à un serveur LogCentral ou se baser sur les traces d'exécution obtenues par DietLogTool. C'est de cette dernière façon que nous avons utilisé le système.

### 6.5.2 Expérimentations et résultats

Nous avons réalisé deux types d'expérimentations sur des plates-formes identiques. La première est la mise en œuvre de notre algorithme de calcul du placement et de l'ordonnancement par le biais de la résolution d'un programme linéaire. La seconde expérience a pour but d'avoir un temps de comparaison pour pouvoir confirmer, ou non, les performances de notre algorithme. Par défaut, l'ordonnancement de DIET de fait un *round-robin*, c'est-à-dire que le serveur choisi pour traiter une requête est celui, parmi ceux qui disposent du service requis, qui n'a pas reçu de requête depuis le plus longtemps. Dans notre cas, où la gestion des données est primordiale, cela ne pouvait donner des résultats pertinents, les données auraient sans cesse transité d'un serveur à

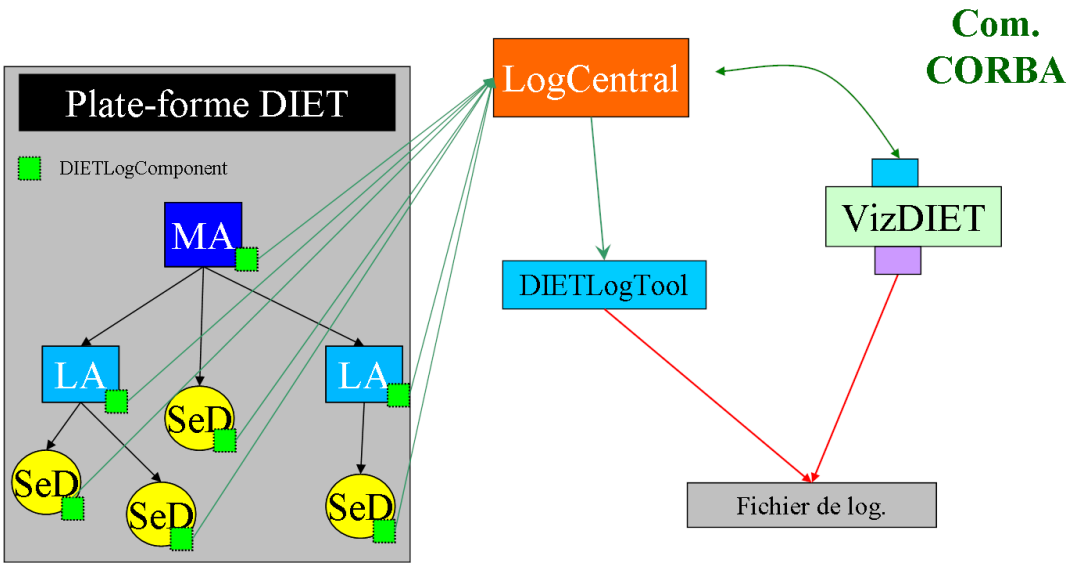


FIG. 6.9 – L'architecture du système de surveillance LogService mis en œuvre dans DIET.

l'autre. Pour cela, nous avons choisi de toujours faire un round-robin pour l'ordonnancement mais également de fixer le placement des données. Ainsi,, une fois en place, celles-ci ne bougeront plus. Pour ce placement, nous avons utilisé l'algorithme glouton déjà présenté dans la section 4.3.5.

Les expérimentations ont eu lieu trois fois pour chacune des deux méthodes testées. Le tableau 6.2 présente les moyennes des résultats de ces expériences.

| | SRA | glouton |
|---|---|---|
| temps de calcul | 41618 s | 47953 s |
| temps moyen par requête | 8,32 s | 9,59 s |
| nb de transfert | 29 | 101 |

TAB. 6.2 – Résultats moyens obtenus lors des expérimentations sur la plate-forme *Grid'5000*.

Les figures 6.11 et 6.12 montrent la plate-forme en fin d'une des expérimentations concernant SRA pour la première figure et une basée sur le placement glouton pour la seconde. Le rectangle au centre est le MA et les ovales l'entourant sont les SeD. Les carrés sur l'extèrieur représentent les données stockées sur chaque SeD. Il est a noté que dans les deux cas, le SeD en bas à droite est celui qui sert de centre de stockage et contient toutes les données au départ de l'expérience. Nous pouvons voir que le nombre de données sur les SeD dans le cas de l'algorithme SRA est bien moindre que dans l'autre cas.

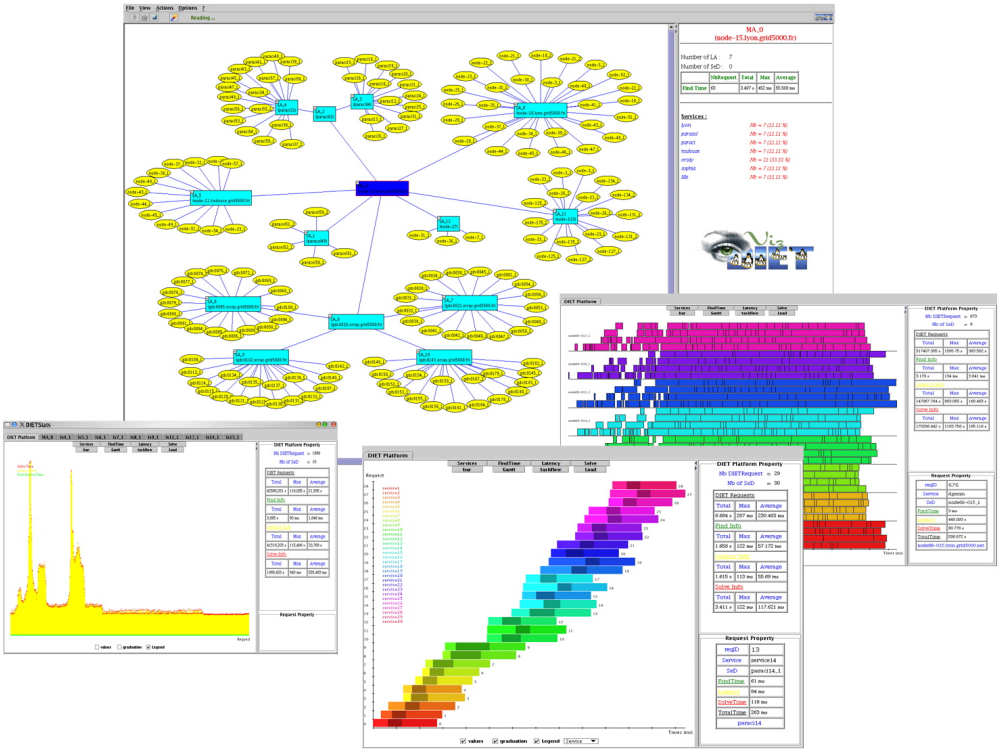

FIG. 6.10 – Aperçu des différentes représentations graphiques offertes par VizDiet.

Les figures 6.13 et 6.14 montrent les diagrammes de Gantt de l'utilisation des SeD au cours de l'une des expérimentations concernant SRA pour la première figure et celle basée sur le placement glouton pour la seconde. Dans ces graphiques, chaque bande représente l'utilisation d'un SeD, la présence de couleur indique son utilisation. Le nœud représenté en bas de ces graphiques est le nœuds de stockage, il est donc normal qu'il n'y est aucune activité durant la phase de calcul. Dans le cas de l'expérimentation de SRA, nous voyons que la majorité des nœuds sont utilisés en quasi permanence. Dans le cas du placement glouton, cette utilisation est nettement plus clairsemée et seuls quatre des dix SeD en présence sont utilisés en quasi permanence laissant les six autres largement sous-utilisés. De cela, plusieurs conclusions : la première est que l'excès de données est inutile et que dans ce cas notre algorithme SRA évite un grand nombre de transferts inutiles et surtout une multiplication des réplicats des banques de données dont le coût de maintien à jour peut vite devenir prohibitif. La seconde conclusion serait que même dans le cas où l'espace de stockage permet de placer un grand nombre de banques de données sur l'ensemble des nœuds, l'ordonnancement des requêtes reste un enjeu majeur. En effet, malgré un grande nombre de replicats des données les plus consommatrices en volume de calcul, le résultat obtenu avec un placement glouton et un ordonnancement round robin montre que ces réplicats de

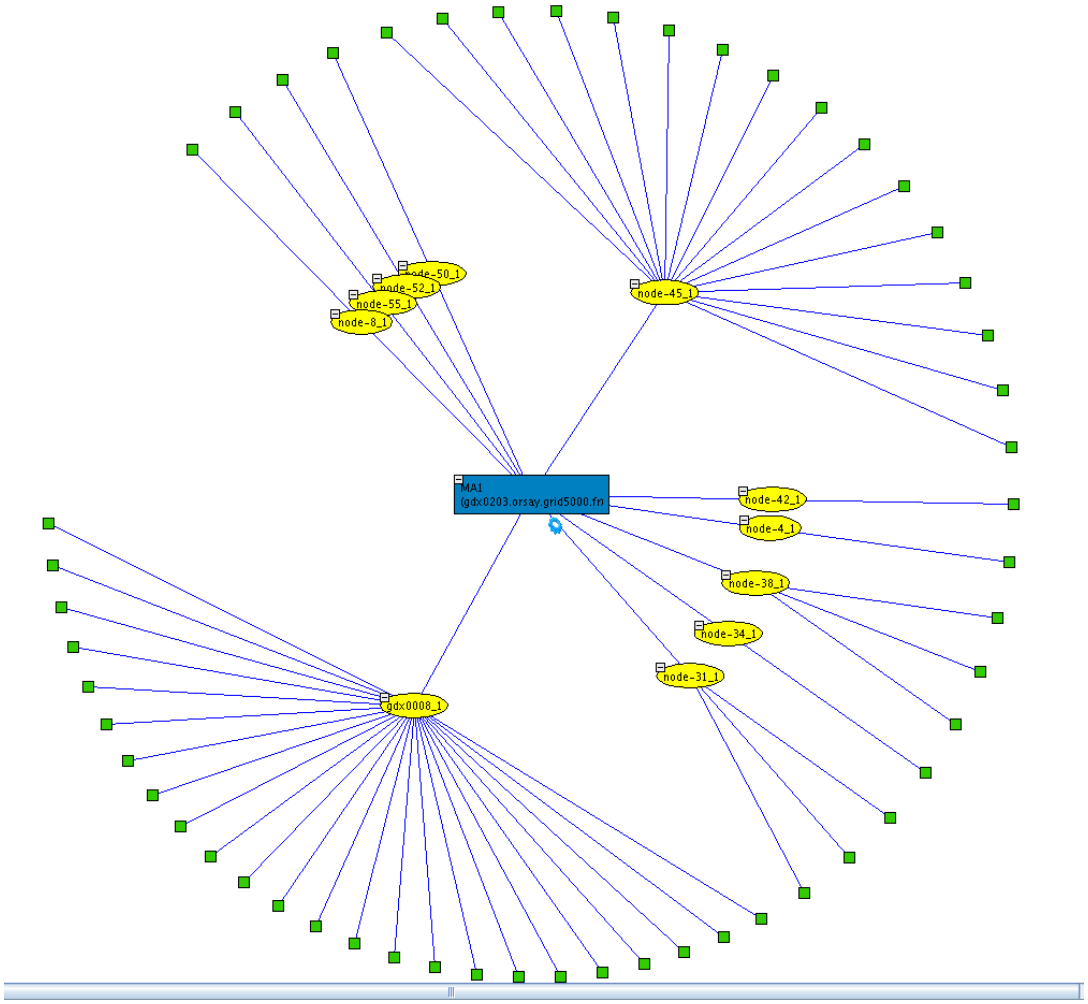


FIG. 6.11 – Plate-forme et répartition des données à la fin de l'expérimentation sur *Grid'5000* de l'algorithme SRA.

sont pas utilisés de façon efficace.

## 6.6 Conclusion

Dans ce chapitre nous avons vu comment il était possible d'intégrer notre algorithme statique simplement au sein d'un intergiciel de grille déjà existant. Si un intergiciel possède un gestionnaire de données avec une interface permettant la manipulation des données au niveau du gestionnaire de ressource, et s'il est possible d'accéder ou de modifier l'ordonnanceur, alors notre solution peut facilement être mise en place dans un tel système. D'autre part, les expérimentations sur la plate-forme *Grid'5000* nous ont permis de valider le résultat des performances obtenues par notre algorithme

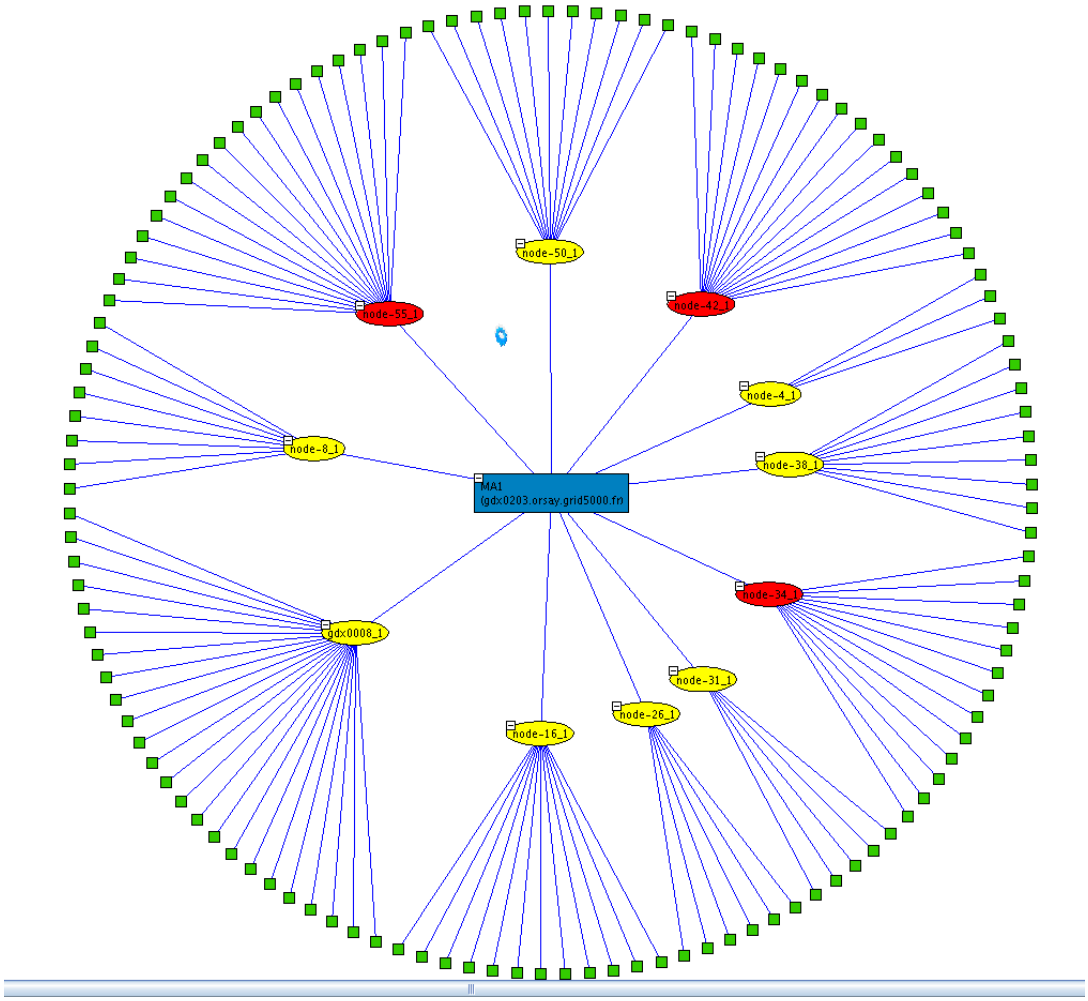


FIG. 6.12 – Plate-forme et répartition des données à la fin de l'expérimentation sur *Grid'5000* avec le placement glouton.

face à une méthode simple mais basée sur un placement qui a fait ses preuves lors des étapes de simulations.

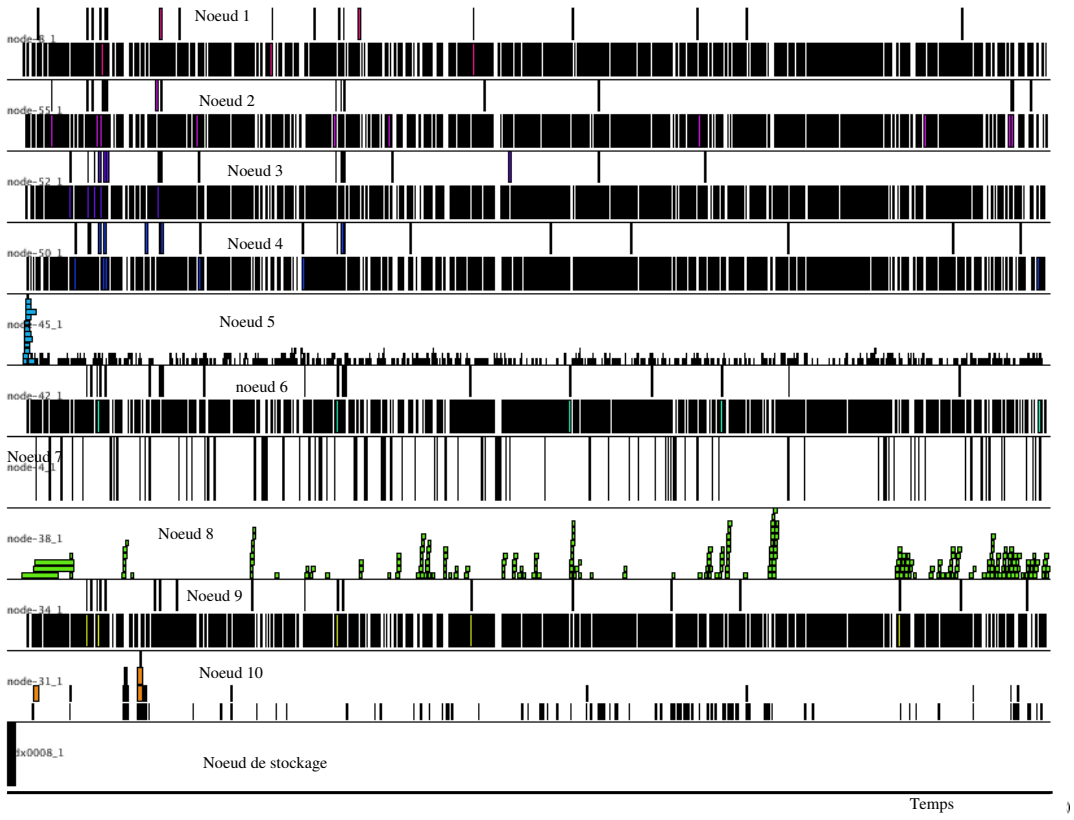


FIG. 6.13 – Diagramme de Gantt de l'utilisation des différents SeD au cours du temps durant de l'expérimentation de l'algorithme SRA.

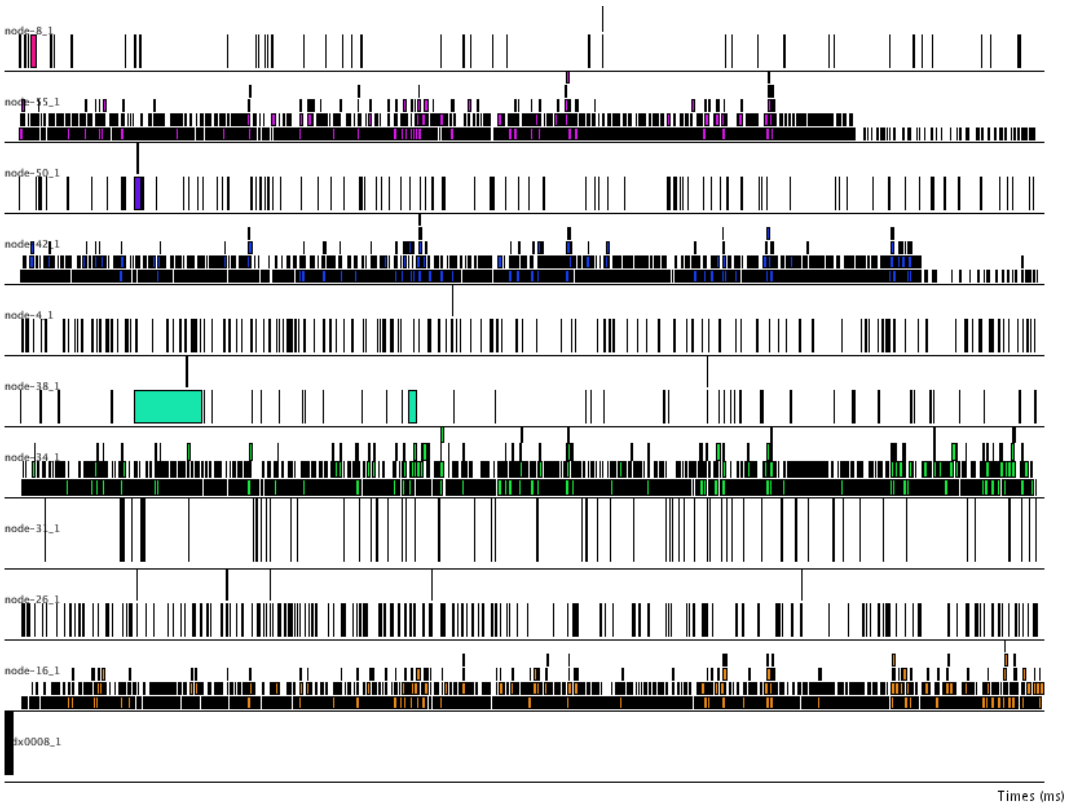


FIG. 6.14 – Diagramme de Gantt de l'utilisation des différents SeD au cours du temps durant de l'expérimentation avec le placement glouton.

# Chapitre 7

# Conclusions

## 7.1 Contributions

Au cours de cette thèse, nous nous sommes placé dans le contexte bien particulier d'une catégorie d'applications bioinformatique dont les caractéristiques sont d'utiliser des banques de données de référence en lecture seule et d'avoir un coût en temps de calcul affine en la taille des données. Une autre caractéristique concernant l'utilisation de ces applications est que leur schéma d'utilisation reste constant dans le temps. Dans ce cadre, nous avons étudié le problème du placement des données et de l'ordonnancement des tâches.

**Algorihmes d'ordonnancement et réplication simultanés** Nous avons défini un premier algorithme statique basé sur un programme linéaire permettant de calculer un ordonnancement et un placement des données optimisant le rendement d'une plate-forme de type grille de calcul. Partant de cette version statique, nous avons établi un algorithme dynamique pour rétablir l'équilibre des charges entres les serveurs en cas de variations entre la réalite des requêtes soumises et les caractéristiques du flot de requêtes utilisées pour réaliser le placement initial. Pour cela, nous avons établi différentes heuristiques. Les simulations nous ont permis d'exhiber le comportement de ces heuristiques en fonction de divers paramètres et de montrer qu'il est possible, malgré de fortes fluctuations, de conserver une utilisation efficace de la plate-forme.

**Simulation** Pour tester nos algorithmes et heuristiques nous avons largement modifié un simulateur de grille existant : Optorsim. Le résultat est un simulateur utilisant les mécanismes de base d'Optorsim mais permettant une gestion plus souple et plus complexe des tâches, de leur exécution. La gestion des techniques d'ordonnancement et du placement des données ont également été rendu plus ouvert afin de pouvoir facilement intégrer, et tester, de nouveaux algorithmes.

**Implémentation** Enfin, nous avons réalisé un prototype du système basé sur l'ordonnancement et le placement statique au sein de l'intergiciel de grille DIET. Ce prototype déployé sur un ensemble de nœuds de la plate-forme *Grid 5000* nous a permis de montrer la possibilité d'intégrer facilement le type d'algorithmes que nous proposons au sein d'un intergiciel de grille. Il a également validé les résultats concernant l'efficacité de l'algorithme statique obtenus par simulations.

## 7.2 Perspectives

Nos travaux ont permis de montrer l'efficacité d'une approche intégrant le calcul du placement des données à l'ordonnancement dans le contexte de la bioinformatique. Mais il s'agit d'un début pour les recherches dans le domaine de l'ordonnancement conjoint des données et des calculs qui ouvrent des perspectives directes aussi bien d'un point de vue pratique que théorique ou encore algorithmique.

Tout d'abord, il serait intéressant d'implémenter les heuristiques dynamiques dans un intergiciel existant disposant d'un système d'information permettant de collecter les données nécessaires à l'adaptation dynamique. Le couple DIET et DTM a déjà montré qu'il se prêtait parfaitement à l'implémentation de l'algorithme dynamique. Adjoint à CORI, le nouveau mécanisme de remonter d'informations sur l'état de plate-forme DIET, DIET et DTM pourraient encore être un intergiciel offrant les mécanismes nécessaires pour intégrer les techniques d'adaptation dynamique du placement et de l'ordonnancement.

D'un point de vue théorique et algorithmique, pour établir l'algorithme dynamique, nous n'avons considéré que des différences entre le schéma d'utilisation des requêtes utilisé pour le premier ordonnancement et les fréquences d'apparition des requêtes réellement constatées. Il pourrait être intéressant d'étudier le comportement des heuristiques dynamiques dans le cas où ce sont les performances de la plate-forme qui ne sont pas celle prévue comme la défaillance de nœuds ou l'arrivée, ponctuelle ou non, de charges provenant de tâches extérieures sur les nœuds de calcul. De même, pour ces heuristiques dynamiques, nous avons opté pour une approche raffinant successivement le placement et l'ordonnancement afin de palier aux différences entre le schéma d'utilisation prévu et la réalité. À chaque itération, nous ne modifions que le placement d'une donnée et adaptons l'ordonnancement en conséquences. Nous pourrions alors nous intéresser à une approche plus globale en cherchant à remettre en cause l'intégralité du placement et de l'ordonnancement. Cette méthode soulève alors d'autres points de recherche comme les méthodes nécessaires pour passer d'un placement de données à un autre ou des métriques pour évaluer si le coût de changer de placement est rentable par rapport aux performances actuelles de la plate-forme. L'idée d'une approche globale est d'ailleurs une des bases de la thèse de Gaël Le Mahec débutée en septembre 2005 conjointement avec le Laboratoire de l'Informatique du Parallélisme et le Laboratoire de Physique Corpusculaire.

Plus généralement, considérer simultanément l'ordonnancement des tâches et le placement des données est une approche relativement récente qui semblent offrir une réponse pour l'utilisation efficace des ressources d'une grille de calcul. Avec nos hypothèses concernant les applications et l'utilisation des données complètes et simples, tout en étant réalistes, le problème du placement et de l'ordonnancement conjoint reste NP-complet et donc complexe à résoudre. D'ailleurs, selon [12], le placement des données est à lui seul un problème NP-complet. Cela laisse à croire, qu'une solution dans un cadre plus général restera complexe à résoudre. Malgré l'augmentation des débits réseaux et la diminution des coûts de stockage, la taille sans cesse croissante des données bioinformatiques, mais aussi d'autres domaines comme la physique ou l'océanographie, font du placement des données en vue de leur utilisation un point de plus en plus important pour utiliser efficacement des environnements de grilles de calculs. Puisqu'il s'agit essentiellement d'effectuer des calculs sur ces données, l'ordonnancement est également important. Les bons résultats que nous avons obtenus, particulièrement ceux de l'algorithme d'approximation et de certaines heuristiques du cadre dynamique, permettent de penser que, même avec des hypothèses moins strictes, il est possible d'utiliser efficacement une plate-forme en optimisant placement et ordonnancement.

# Chapitre 8

# Bibliographie


[1] W. Bell, D. Cameron, L. Capozza, A. Millar, K. Stockinger et F. Zini. « OptorSim - A Grid Simulator for Studying Dynamic Data Replication Strategies ». *International Journal of High Performance Computing Applications* **17(4)** (2003).

[2] W. Bell, D. Cameron, L. Capozza, A. Millar, K. Stockinger et F. Zini. « Simulation of Dynamic Grid Replication Strategies in OptorSim ». Dans *Proc. of the 3rd Int'l. IEEE Workshop on Grid Computing (Grid'2002)* (Baltimore, USA, novembre 2002), Lecture Notes in Computer Science, Springer Verlag.

[3] W. Bell, D. Cameron, R. Carvajal-Schiaffino, A. Millar, K. Stockinger et F. Zini. « Evaluation of an Economy-Based File Replication Strategy in Data-Grids ». Dans *Third Internation Symposium on Cluster Computing and the Grid (CCGRID)* (2003).

[4] B. Boeckmann, A. Bairoch, R. Apweiler, M.-C. Blatter, A. Estreicher, E. Gasteiger, M. Martin, K. Michoud, C. O'Donovan, I. Phan, S. Pilbout et M. Schneider. « The SWISS-PROT Protein Knowledgebase and its Supplement TrEMBL in 2003 ». *Nucleic Acids Res.* **31** (2003), 365–370.

[5] R. Bolze, E. Caron et F. Desprez. « A Monitoring and Visualization Tool and Its Application for a Network Enabled Server Platform ». Dans *Parallel and Distributed Computing Workshop of ICCSA 2006* (Glasgow, UK., 8-11 May 2006), LNCS, éditeur. To appear.

[6] P. Bucher et A. Bairoch. « A Generalized Profile Syntax for Biomolecular Sequences Motifs and Its Function in Automatic Sequence Interpretation ». Dans *Proceedings 2nd International Conference on Intelligent Systems for Molecular Biology* (1994), R. Altman, D. Brutlag, P. Karp, R. Lathrop et D. Searls, éditeurs, volume 2, AAAIPress, pp. 53–61.

[7] N. Capit, G. D. Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron et O. Richard. « A batch scheduler with high level components ». Dans *Cluster computing and Grid 2005 (CCGrid05)* (2005).

[8] E. Caron, P. K. Chouhan et H. Dail. « GoDIET : A Deployment Tool for Distributed Middleware on Grid'5000 ». Dans *EXPGRID workshop. Experimental Grid*

*Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In conjunction with HPDC-15.* (Paris, France, June 19th 2006), IEEE, éditeur, pp. 1–8.

[9] E. Caron et F. Desprez. « DIET : A Scalable Toolbox to Build Network Enabled Servers on the Grid ». *International Journal of High Performance Computing Applications* (2006). to appear.

[10] H. Casanova, A. Legrand et L. Marchal. « Scheduling Distributed Applications : the SimGrid Simulation Framework ». Dans *Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)* (may 2003), IEEE Computer Society Press.

[11] A. Chakrabarti, R. Dheepak et S. Sengupta. « Integration of Scheduling and Replication in Data Grids ». Dans *Proceedings 11th International Conference on High Performance Computing (HiPC 2004)* (Bangalore, India, décembre 2004), L. Bougé et V. K. Prasanna, éditeurs, Springer, pp. 375–385.

[12] U. Cibej, B. Slivnik et B. Robic. « The Complexity of Static Data Replication in Data Grids ». *Parallel Computing* **31** (2005), 900–912.

[13] I. Clarke, O. Sandberg, B. Wiley et T. Hong. « Freenet : A Distributed Anonymous Information Storage and Retrieval System ». Dans *? ? ?* (2001), Numéro 2009 dans Lecture Notes in Computer Science.

[14] C. Combet, C. Blanchet, C. Gourgeon et G. Deléage. « NPS@ : Network Protein Sequence Analysis ». *TIBS* **25, No 3** (mars 2000), [291] :147–150.

[15] D. Coudert et H. Rivano. « Lightpath assignment for multifibers WDM optical networks with wavelength translators ». Dans *IEEE Global Telecommunications Conference (Globecom'02)* (2002), IEEE Computer Society Press. Session OPNT-01-5.

[16] F. Desprez, M. Quinson et F. Suter. « Dynamic Performance Forecasting for Network-Enabled Servers in a Heterogeneous Environment ». Dans *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (Las Vegas, juin 2001), H. Arabnia, éditeur, volume III, CSREA Press, pp. 1421–1427. ISBN : 1-892512-69-6.

[17] I. Foster. « Globus Toolkit Version 4 : Software for Service-Oriented Systems ». Dans *Proceedings of the International Conference on Network and Parallel Computing* (2005), Lecture Notes in Computer Science, Springer-Verlag, pp. 2–13.

[18] I. FOSTER ET C. KESSELMAN, éditeurs. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[19] GRIPPS. http://gripps.ibcp.fr/index.php.

[20] M. R. Garey et D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[21] INFOBIOGEN. « Introduction à la bioinformatique ». http://www.infobiogen.fr/doc/documents.php?cours=bioinfo_intro.

[22] Institut de Biologie et Chime des Protéines. http://www.ibcp.fr.

[23] K. Calvert and M. Doar and E.W. Zegura. « Modeling Internet Topology ». *IEEE Communications Magazine* **35** (1997), 160–163.

[24] G. Kan. *Peer-to-Peer : Harnessing the Power of a Disruptive Technology*. O'Reilly, 2001, chapitre Gnutella, pp. 94–122.

[25] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells et B. Zhao. « OceanStore : An Architecture for Global-scale Persistent Storage ». Dans *Proceedings of ACM ASPLOS* (November 2000), ACM.

[26] Le projet Grid'5000. http://www.grid5000.fr.

[27] A. Legrand, A. Su et F. Vivien. « Off-line scheduling of divisible requests on an heterogeneous collection of databanks ». Dans *Proceedings of the 14th Heterogeneous Computing Workshop* (Denver, Colorado, USA, avril4 2005), IEEE Computer Society Press, p. (10 pages).

[28] LogService. http://graal.ens-lyon.fr/DIET/logservice.html.

[29] NCBI. « Just the Facts : A Basic Introduction to the Science Underlying NCBI Resources ». http://www.ncbi.nlm.nih.gov/About/primer/bioinformatics.html.

[30] NPS@ : Network Protein Sequence Analysis. http://npsa-pbil.ibcp.fr.

[31] S.-M. Park, J.-H. Kim, Y.-B. Ko et W.-S. Yoon. « Dynamic Data Grid Replication Strategy Based on Internet Hierarchy. ». Dans *GCC (2)* (2003), pp. 838–846.

[32] G. Peng. « CDN : Content Distribution Network ». Rapport technique TR-125, Experimental Computer Systems Lab, Department of Computer Science, State University of New York, Stony Brook, 2003.

[33] S. Podlipding et L. Böszörmenyi. « A Survey of Web Cache Replacement Strategies ». *ACM Computing Surveys* **35**, numéro 4 (décembre 2003), 374–398.

[34] K. Ranganathan et I. Foster. « Identifying Dynamic Replication Strategies for a High Performance Data Grid ». Dans *Proc. of the Second International Workshop on Grid Computing* (2001).

[35] K. Ranganathan et I. Foster. « Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids ». *Journal of Grid Computing* **1**, numéro 2 (avril 2003), 53–62.

[36] Renater : Réseau National de télécommunications pour la Technologie l'Enseignement et la Recherche. http://www.renater.fr.

[37] N. S. Rich Wolski et J. Haye. « The Network Weather Service : A Distributed Resource Performance Forecasting Service for Metacomputing ». *Journal of Future Generation Computing Systems* **15** (octobre 1999), 757–768.

[38] A. I. T. Rowstron et P. Druschel. « Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility ». Dans *Symposium on Operating Systems Principles* (2001), pp. 188–201.

[39] O. Soyez. « Us : prototype de stockage pair à pair ». Dans *15e Rencontres Francophones en Parallélisme, La Colle sur Loup, France* (2003), M. Auguin, F. Baude, D. Lavenier et M. Riveill, éditeurs.

[40] The European DataGrid Project. http://www.eu-datagrid.org.

[41] S. Venugopal, R. Buyya et K. Ramamohanarao. « A Taxonomy of Data Grids for Distributed Data Sharing, Management and Processing ». *ArXiv Computer Science e-prints* (juin 2005).

[42] J. Wang. « A survey of web caching schemes for the Internet ». *SIGCOMM Comput. Commun. Rev.* **29**, numéro 5 (1999), 36–46.

[43] O. Wolfson, S. Jajodia et Y. Huang. « An adaptive data replication algorithm ». *ACM Trans. Database Syst.* **22**, numéro 2 (1997), 255–314.

[44] C. Wu, L. Yeh, H. Huang, L. Arminski, J. Castro-Alvear, Y. Chen, Z. Hu, P. Kourtesis, R. Ledley et B. e. a. Suzek. « The Protein Information Resource ». *Nucleic Acids Res.* **31** (2003), 345–347.

# Chapitre 9

# Liste des publications

## Articles parus dans des journaux internationaux

[45] F. Desprez et A. Vernois. « Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid ». *Journal Of Grid Computing* **4**, numéro 1 (mars 2006), 19–31.

## Actes de conférences internationales avec comité de lecture

[46] G. Utard et A. Vernois. « Data Durability in Peer-to-Peer Storage Systems ». Dans *Proc. 4th Workshop on Global and Peer to Peer Computing* (Chicago, avril 2004), IEEE/ACM CCGrid Conference, p. (9 pages).

[47] F. Desprez et A. Vernois. « Simultaneous Scheduling of Replication and Computation on the Grid ». Dans *CLADE 2005* (Research Triangle Park, NC, juillet 2005), IEEE Computer Society Press.

[48] C. Blanchet, F. Desprez et A. Vernois. « Simultaneous Scheduling of Replication and Computation for Bioinformatic Applications on the Grid ». Dans *Biological and Medical Data Analysis, Proceedings of 6th International Symposium, ISBMDA 2005* (Aveiro, Portugal, novembre 2005), volume 3745 / 2005, Springer-Verlag GmbH.

## Articles parus dans des journaux francophones

[49] A. Vernois. « Pérennité des données dans les systèmes de stockage pair-à-pair ». *Technique et Science Informatique (TSI), Numéro spécial RenPar'14* (A paraître).

## Actes de conférences nationales avec comité de lecture

[50] A. Vernois. « Pérennité dans les systèmes de stockage pair à pair ». Dans *15e Rencontres Francophones en Parallélisme, La Colle sur Loup, France* (octobre 2003), M. Auguin, F. Baude, D. Lavenier et M. Riveill, éditeurs, pp. 153–160.

[51] A. Vernois. « Ordonnancement conjoint du placement des données et des calculs sur la grille ». Dans *16e Rencontres Francophones en Parallélisme, Le Croisic, France* (avril 2005).

## Actes de conférences nationales sans comité de lecture

[52] E. Caron, F. Desprez, B. Del-Fabbro et A. Vernois. « Gestion de données dans les NES ». Dans *DistRibUtIon de Données à grande Echelle. DRUIDE 2004* (Domaine du Port-aux-Rocs, Le Croisic. France, mai 2004), IRISA, pp. 23–32.

## Rapports de recherche

[53] F. Desprez et A. Vernois. « Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid ». Rapport technique RR2005-01, LIP, jan 2005.

**Résumé :**
Au cours de cette thèse, nous nous sommes placés dans le contexte d'une catégorie d'applications bioinformatiques dont les caractéristiques sont d'utiliser des banques de données de références en lecture seule et d'avoir un coût en temps de calcul affine en la taille des données. Une autre caractéristique concernant l'utilisation de ces applications est que leur schéma d'utilisation reste constant dans le temps. Dans ce cadre, nous avons défini un algorithme basé sur un programme linéaire permettant de calculer un ordonnancement et un placement statique des données optimisant le rendement d'une plate-forme de type grille de calcul. Grâce au simulateur Optorsim que nous avons largement modifié, nous avons montré les bons résultats de notre algorithme lorsque l'espace de stockage sur les nœuds de calcul ou le débit du réseau connectant les différents sites sont des points critiques. Nous avons ensuite établi un ensemble d'heuristiques dont le but est de palier à d'éventuels changements dans les schémas d'utilisation des données. Là encore, nous avons utilisé Optorsim pour montrer et comprendre l'impact de ces différentes heuristiques. Il en découle que dans la plupart des cas, nous sommes en mesure de conserver une utilisation presque optimale de la plate-forme. Enfin, nous avons réalisé un prototype de l'algorithme statique au sein de l'intergiciel de grille DIET. Ce prototype, déployé sur des nœuds de la plate-forme Grid 5000, nous a permis de montrer l'efficacité de notre méthode dans un environnement réel.

**Mots-clés :**
Ordonnancement, gestion de données, grille de calcul, régime permanent

**Abstract:**
In this thesis, we focus on the specific context of a set of bioinformatics applications. Those applications have the particularity to use well-known read-only databanks and a computation cost linear with the size of data. The other point is that the use of application and data is always the same. Within this context, we have developed an algorithm that combines data management and scheduling using steady-state approach. Using a model of the platform, the number of requests, as well as their distribution, the number and size of databanks, we define a linear program to satisfy all constraints at every level of the platform. The solution of this linear program gives us a scheduling for the request and a static placement of databanks on servers. Using the Optorsim grid simulator, we have show good results when storage space or network bandwidth between grid nodes are critical resources. Then, we have designed a set of heuristics that aim to adapt scheduling and data placement when data use changes. These heuristics have also been tested with OptorSim. The conclusion of these simulations is that, in most cases, our algorithm is able to keep an almost optimal use of computation resources. Finally, we have implemented a prototype of our static solution inside the DIET grid middleware. This prototype has been deployed on the Grid'5000 platform. The run of experiments in these real conditions has validated results of our simulations.

**Keywords:**
Scheduling, data management, grid computing, steady state.