Exécution d'applications parallèles en environnements hétérogènes et volatils : déploiement et virtualisation

Sébastien Miquée

Exécution d'applications parallèles en environnements hétérogènes et volatils : déploiement et virtualisation

Thèse pour obtenir le grade de
Docteur de l'Université de Franche-Comté
Spécialité Informatique Parallèle et Distribuée

par

Sébastien Miquée

Laboratoire Femto-ST, Département Informatique des Systèmes Complexes
UFR Sciences et Techniques
Université de Franche-Comté

Soutenue le 25 janvier 2012 devant la commission d'examen :

Rapporteurs	**Christophe Cérin**	Professeur à l'Université Paris XIII
	Stéphane Genaud	Maître de Conférences HDR à l'Université de Strasbourg
Examinateurs	**Nabil Abdennadher**	Professeur à l'Université des Sciences Appliquées de Suisse Occidentale
	Laurent Philippe	Professeur à l'Université de Franche-Comté
Directeur	**Raphaël Couturier**	Professeur à l'Université de Franche-Comté
Co-encadrant	**David Laiymani**	Maître de Conférences à l'Université de Franche-Comté

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Remerciements

En premier lieu je tiens à remercier tout particulièrement mes parents, qui m'ont aidé et encouragé tout au long de mes études et plus particulièrement ces trois dernières années.

Je tiens également à remercier Jacques pour m'avoir accueilli au sein de l'équipe, et Raphaël et David de m'avoir fait confiance et de m'avoir accepté en thèse, après m'avoir eu comme étudiant en DUT. J'ai notamment apprécié leur sérieux et leur rigueur dans le travail, ainsi que l'aide précieuse qu'ils m'ont apportée tout au long de cette thèse. J'ai aussi apprécié les moments que l'on a passés en dehors du travail, et surtout leur compréhension dans les moments difficiles.

Je tiens à remercier messieurs Christophe Cérin et Stéphane Genaud pour avoir accepté d'être les rapporteurs de cette thèse. Je remercie également messieurs Nabil Abdennadher et Laurent Philippe d'avoir accepté de faire partie des membres du jury.

J'adresse des remerciements tout particuliers aux membres de l'équipe AND pour leur accueil et les bons moments que nous avons passés. Je pense notamment à Lilia qui a dû me supporter ces deux dernières années, à Arnaud pour nos pauses toujours constructives, à Jean-Luc pour les cours de système et réseau durant lesquels nous avons « torturé » les étudiants avec des problèmes complexes, à Michel et Gilles pour leur bonne humeur, à Mourad, Abdallah et Jean-Claude pour nos sorties extra-universitaires, à Christophe pour sa précieuse aide mathématique et nos discussions sur des sujets atypiques, à Nicolas pour les moments latinos, à Karine, Patricia, Stéphane, Jeff et Fabrice qui ont aussi contribué à la bonne humeur ambiante. J'ai également une pensée pour Babeth, Laurence et Husam, qui sont partis vers des contrées lointaines. J'adresse un grand merci à toutes ces personnes.

Je tiens également à remercier toutes les personnes que j'ai pu rencontrer au fil des conférences et des réunions de travail, pour les échanges d'idées, souvent constructives, qui ont permis d'explorer de nouvelles pistes de recherche.

Je tiens également à remercier le projet *InterReg IV From-P2P* pour avoir financé cette thèse, ainsi que Grid'5000 et le département Informatique de l'IUT de Belfort-Montbéliard pour nous avoir fourni les plateformes d'exécution pour toutes nos expérimentations.

La connaissance scientifique possède en quelque sorte des propriétés fractales : nous aurons beau accroître notre savoir, le reste – si infime soit-il – sera toujours aussi infiniment complexe que l'ensemble de départ.

Isaac Asimov

Introduction

Depuis très longtemps, l'analyse numérique est utilisée par des mathématiciens afin de résoudre des problèmes difficiles et longs. Ce domaine théorique a permis de construire des méthodes de traitement encore utilisées aujourd'hui, comme l'élimination de Gauss ou la méthode de Newton. L'analyse numérique a pour objectif la création d'algorithmes permettant de résoudre des problèmes de mathématiques à variables continues. Son principal champ d'action est la résolution de problèmes à variables réelles ou complexes, comme par exemple l'algèbre linéaire numérique sur les champs réels ou complexes, la recherche de solutions numériques d'équations différentielles ... Il existe principalement deux familles d'algorithmes : les méthodes directes et les méthodes itératives. Les premières résolvent le problème en un nombre fini d'opérations et donnent un résultat exact, alors que les secondes répètent le même bloc d'instructions jusqu'à ce qu'un seuil de précision donné soit atteint. Les méthodes itératives donnent un résultat approché de la solution et sont pour certains problèmes les seules méthodes efficaces, comme lorsque les données à traiter sont trop volumineuses ou pour certains types de calcul (comme le calcul des racines d'un polynôme par exemple). De nos jours, la partie pratique de cette analyse numérique, c'est-à-dire la réalisation des calculs, est traitée par des ordinateurs, permettant aux scientifiques et aux industriels de progresser dans leurs recherches. Les bénéfices de ces calculs permettent par exemple de prédire plus finement et à plus long terme la météorologie, de construire des véhicules ou des édifices plus sûrs grâce à l'étude de la résistance des matériaux, ou encore de cibler plus finement une zone de cancer et de délivrer une dose plus adaptée en radiothérapie.

Dans ce contexte, les nouvelles technologies disponibles permettent aux scientifiques de divers domaines, tels que la biologie, la simulation climatologique ou la physique, d'obtenir des données de plus en plus précises et volumineuses. Bien que les machines actuelles offrent de plus en plus de ressources, avec de nombreux cœurs de calcul cadencés à des fréquences toujours plus élevées, des mémoires plus rapides et plus volumineuses, celles-ci ne suffisent plus pour résoudre des problèmes de si grande taille. Au cours des dernières décennies, de nombreuses architectures de calcul ont été mises en place afin de permettre l'exécution d'applications scientifiques pour résoudre ces problèmes de grande taille. Tout d'abord sont apparus les supercalculateurs, proposant en une seule « machine » la puissance de plusieurs, tout en offrant des liens de communication performants, car internes à la machine. Bien que performants, les supercalculateurs ont le principal inconvénient d'être onéreux, tant en terme d'investissement qu'en coût de maintenance. La principale raison de leur coût élevé est l'utilisation de processeurs spécifiques à cette architecture. La tendance actuelle est plus à l'utilisation de grappes de machines, ou clusters. Les machines classiques devenant plus puissantes et bon marché, cette architecture permet de les rassembler afin de mutualiser leurs ressources. Ceci permet de disposer d'une grande puissance de calcul à faible coût. Néanmoins, devant un besoin croissant de puissance et de ressources, une nouvelle architecture est apparue. Cette solution consiste non pas à agrandir les clusters, mais à

les fédérer au sein d'une architecture plus grande qu'est la grille de calcul. Cette architecture permet de rassembler les ressources de différentes institutions, afin d'augmenter les ressources disponibles. Ces ressources, telles que des clusters ou des supercalculateurs, sont reliées entre elles à l'aide de réseaux haute performance.

Afin d'utiliser ces architectures massivement parallèles, les applications de calcul et par extension les algorithmes sous-jacents, ont dû être adaptés, c'est-à-dire parallélisés. Il existe plusieurs modèles de parallélisation, chacun étant plus ou moins destiné à un type d'application. Le principe de la parallélisation d'une application est de découper ses calculs en plusieurs tâches de calcul, travaillant chacune sur un sous-ensemble des données initiales. On peut classer les applications, ou algorithmes parallèles, en deux grandes familles : les applications à traitements isolés et celles communiquantes. Pour les premières, les calculs effectués par chaque tâche sont indépendants de ceux effectués par les autres tâches. Par exemple, l'exploration d'un arbre peut être réalisée par des tâches parcourant chacune une branche de celui-ci. Seules une étape d'initialisation et une étape de collecte des résultats doivent être réalisées en supplément des calculs. Pour la seconde famille d'applications, celles communiquantes, des bibliothèques de fonctions offrant des possibilités de communication sont utilisées. Par exemple, les bibliothèques mettant en œuvre la norme MPI permettent aux tâches des applications de s'échanger des données, appelées dépendances, à l'aide d'envois et de réceptions de messages. Comme nous l'avons énoncé précédemment, il existe deux grandes familles d'algorithmes, les méthodes directes et les méthodes itératives. Les secondes sont plus facilement parallélisables que les premières, et sont généralement plus efficaces pour le traitement des problèmes comportant des données volumineuses. Les méthodes itératives peuvent se paralléliser, en utilisant le paradigme de l'échange de messages, suivant deux modes : le mode synchrone et le mode asynchrone. Le premier mode impose que les itérations soient synchrones, c'est-à-dire que toutes les tâches calculent la même itération au même moment, puis à la fin de chaque itération se déroule une phase d'échange des dépendances. Ainsi, dans un environnement d'exécution hétérogène (avec des machines de puissances différentes), l'exécution de tels algorithmes résulte en des temps d'inactivité des processeurs les plus puissants qui doivent attendre ceux étant les moins rapides. Le mode asynchrone permet quant à lui de s'affranchir de ces temps d'attente en permettant de recouvrir les temps de communication par du calcul. Le principe de ce modèle est qu'une tâche effectue le calcul de son itération puis envoie à ses voisins de calcul leurs dépendances et commence immédiatement le calcul de l'itération suivante, sans attendre de recevoir ses dépendances. Ainsi ce modèle permet d'utiliser tout le potentiel de calcul d'une machine, et il est également tolérant aux pertes de messages. Il est à noter que malgré ses caractéristiques, le modèle itératif asynchrone converge plus rapidement si les tâches reçoivent régulièrement des messages de dépendances. Les principaux inconvénients de ce modèle sont que tous les algorithmes itératifs ne peuvent pas être parallélisés suivant ce modèle pour des raisons de convergence, et que le nombre d'itérations permettant d'atteindre la convergence est imprévisible et est généralement supérieur à celui du modèle synchrone.

Les méthodes itératives asynchrones ont montré leur bonne adaptabilité et de bonnes performances dans des environnements d'exécution distribués et volatils, comme des grilles de calcul. Pour la mise en œuvre d'applications reposant sur ce modèle algorithmique, il n'existe à notre connaissance qu'une seule plateforme de développement et d'exécution : JaceP2P-V2. Cette plateforme, totalement décentralisée et tolérante aux pannes, fournit les outils nécessaires au développement de ces applications en mettant à disposition des mécanismes de communications asynchrones et de tolérance aux pannes. Ces derniers reposent sur un modèle de sauvegarde/restauration, dont le principe est d'effectuer périodiquement une sauvegarde de l'état des tâches de calcul et d'envoyer celle-ci à des machines voisines, participantes au calcul. Lorsqu'une machine tombe en panne, une remplaçante est choisie et celle-ci récupère la dernière sauvegarde de la tâche dont la machine vient de tomber en panne, et reprend le calcul. Bien que complète, il manque cependant un élément essentiel à cette plateforme : une politique efficace de placement des tâches de calcul. En effet, de par leur constitution, les grilles de calcul fédèrent de

nombreuses machines qui peuvent être géographiquement éloignées. Le nombre important de machines induit des probabilités de pannes plus élevées et les grandes distances entre les machines induisent des latences provoquant des retards lors des envois de messages de dépendances. Le modèle itératif asynchrone étant par nature tolérant à la perte de messages et ne nécessitant aucune synchronisation, il n'est pas bloqué lorsque des messages arrivent en retard ou lorsque des machines tombent en panne. Cependant, comme nous l'avons indiqué précédemment, ce modèle progresse plus rapidement lorsque chaque tâche reçoit régulièrement et rapidement des messages de dépendances. Ainsi, lorsqu'une application itérative asynchrone est exécutée sur une architecture telle qu'une grille de calcul, le choix de l'affectation des tâches de calcul aux machines doit être effectué en tenant compte des points suivants :

- localité des tâches : afin de permettre à l'application de converger plus rapidement, les tâches doivent être proches les unes des autres. Dans une architecture de type grille de calcul, fédérant de nombreuses ressources, les machines exécutant les tâches de l'application doivent être choisies en tenant compte du réseau les reliant entre elles ;
- puissance des machines : il est évident que la puissance des machines joue un rôle important dans la vitesse de progression de l'application. En effet, plus une machine est puissante plus le temps de calcul d'une tâche avec une charge de calcul importante est petit. Ainsi une politique de placement efficace doit choisir les machines de calcul permettant aux tâches d'avoir le temps d'exécution le plus petit possible ;
- caractéristiques de l'application : il est souvent difficile de trouver dans une grille de calcul un cluster contenant des machines puissantes en nombre suffisant pour exécuter toutes les tâches de l'application. En effet, dans une telle architecture, bien souvent les ressources sont partagées entre plusieurs utilisateurs et donc toutes les machines ne sont pas disponibles. Afin d'affiner le choix des machines de calcul, il est important de tenir compte des caractéristiques de l'application. En effet, si les tâches ont peu de dépendances, elles peuvent être placées sur des machines puissantes (réduction du temps d'exécution) et éloignées (car peu de communications) ; à l'inverse, si les tâches ont de nombreuses dépendances, qui plus est avec des messages de grande taille, il est préférable de choisir des machines proches en terme de réseau (accroissement de la vitesse de convergence globale) même si celles-ci ne sont pas les plus puissantes disponibles ;
- tolérance aux pannes : comme nous l'avons décrit précédemment, la plateforme JaceP2P-V2 utilise un mécanisme de tolérance aux pannes reposant sur le principe de la sauvegarde/restauration de l'état des tâches. Les sauvegardes étant envoyées aux voisins de calcul, ceux-ci doivent être les plus proches possible, afin de limiter les coûts de transfert des sauvegardes. En effet, ces coûts ralentissent aussi bien le processus de sauvegarde que les communications entre les tâches en cours de calcul en occupant une partie de la bande passante. Un autre point à prendre en compte ici est le fait que sur chaque machine n'est exécutée qu'une seule tâche de calcul. Le fait de placer plusieurs tâches sur une même machine diminue l'efficacité des mécanismes de tolérance aux pannes. En effet, si plusieurs tâches sont placées sur une même machine, si celle-ci tombe en panne, il faut alors trouver une machine remplaçante permettant d'exécuter toutes les tâches, mais le point important est que celle-ci doit rapatrier toutes les sauvegardes et relancer tous les calculs. Ces dernières opérations sont coûteuses en temps et en utilisation de la bande passante, et entraînent une perte de temps d'exécution de ces tâches.

Ainsi, on remarque qu'une politique de placement efficace des tâches de calcul d'une application itérative asynchrone sur une architecture distribuée, hétérogène et volatile doit prendre en compte de nombreux paramètres. Un algorithme de placement doit proposer au moins deux mécanismes : un placement initial satisfaisant les conditions énoncées précédemment et une fonction permettant de choisir les machines de remplacement.

Dans cette thèse nous proposons trois algorithmes dédiés à la résolution de cette problématique. Nous proposons tout d'abord deux algorithmes, FT-FEC et FT-AIAC-QM, qui ont chacun pour objectif l'optimisation des deux premiers points, soit respectivement la localité des tâches pour FT-FEC, et

la puissance des machines pour FT-AIAC-QM. Ces deux algorithmes proposent également des fonctions pour la tolérance aux pannes, permettant de choisir des machines de remplacement en accord avec leur politique de placement. Enfin, le troisième algorithme que nous proposons, MAHEVE, permet quant à lui de tenir compte des deux aspects, que sont la localité des tâches et la puissance des machines. Cet algorithme est conçu spécialement pour l'exécution d'applications itératives asynchrones dans des environnements volatils. En effet, ses politiques de placement initial et de choix des machines remplaçantes pour la tolérance aux pannes sont évolutives et s'adaptent aux caractéristiques de l'architecture d'exécution et à celles de l'application, et ce tout au long de l'exécution de l'application. L'adaptation à l'architecture est basée sur son degré d'hétérogénéité. Celui-ci est calculé en fonction du nombre de machines disponibles au sein de chaque cluster et de leur puissance. Ainsi, lorsque des changements s'opèrent au sein de l'architecture durant l'exécution de l'application, comme le départ et/ou l'arrivée de machines de calcul, la politique de l'algorithme s'adapte à ces changements.

L'usage des plateformes de calcul haute performance, comme les grilles de calcul, est pour le moment bien souvent réservé aux entités comme des laboratoires de recherche ou de grandes entreprises, contribuant aux ressources de celles-ci. Cependant, de nombreuses personnes, principalement des scientifiques comme les physiciens par exemple, ayant des applications à exécuter et dont les données sont volumineuses et les temps de calcul sont longs, n'ont pas accès à de telles plateformes de calcul. Les institutions au sein desquelles elles travaillent, des entreprises, des laboratoires ou des universités par exemple, n'ont pas les moyens et/ou l'envie d'investir dans une architecture dédiée à l'exécution de ces applications. Une des solutions envisageables est l'utilisation du « cloud computing ». Cette nouvelle technique permet d'utiliser des machines de calcul d'autres entités. Bien souvent les utilisateurs de cette technologie n'ont aucun contrôle sur les machines utilisées. En effet, le principe du « cloud computing » est de soumettre une application de calcul avec ses données à un organisme, en indiquant les ressources nécessaires à son exécution. L'utilisateur récupère les résultats une fois l'application terminée, laissant à la plateforme la gestion des machines et le bon déroulement de l'exécution de l'application. Dans de nombreux domaines, l'utilisation d'une telle solution n'est pas envisageable, principalement pour des raisons de confidentialité. Une alternative peut être envisagée, permettant à ces personnes d'utiliser une architecture de calcul leur assurant la confidentialité de leurs données et résultats, mais aussi la terminaison de leurs applications, même si le leur temps d'exécution peut prendre plusieurs jours. Comme nous l'avons déjà souligné, les machines actuelles sont de plus en plus puissantes, et leur utilisation en tant que machines de travail au sein des institutions offre un potentiel non négligeable pour l'exécution d'applications de calcul. En effet, ces machines sont bien souvent sous-exploitées et sont une bonne partie du temps non utilisées, comme par exemple la nuit ou pendant les vacances. Une solution est donc d'utiliser ces machines afin de former une architecture de calcul. Les principaux avantages d'une telle architecture sont le faible coût d'investissement et le contrôle possible des machines.

Dans cette thèse nous proposons la mise en œuvre d'une plateforme de calcul permettant d'utiliser ces ressources disponibles, afin de permettre l'exécution d'applications de calcul. Les objectifs de cette plateforme sont les suivants :

- assurer la confidentialité des données et des résultats, et l'intégrité des calculs des applications. En effet, dans un tel environnement, les machines utilisées pour exécuter les applications de calcul sont des machines de travail, impliquant de ce fait leur utilisation par des utilisateurs « normaux ». Il est important que ces derniers ne puissent pas interagir avec les applications de calcul ;
- assurer la terminaison des applications de calcul. Les temps d'exécution des applications pouvant être relativement longs et comme les machines ne leur sont pas dédiées, celles-ci peuvent être trop chargées ou tomber en panne. Une politique de tolérance aux pannes doit être présente afin d'éviter l'arrêt de l'application ;

- être facile d'utilisation. Une telle plateforme est destinée à des non informaticiens, et par conséquent son installation, sa configuration et son utilisation doivent être simples. Une telle plateforme doit requérir le moins de modifications possibles dans les codes des applications. La gestion de la plateforme doit être la plus transparente possible.

Afin de mettre en œuvre ces objectifs, nous avons opté pour l'utilisation du langage multiplateforme Java en conjonction avec un hyperviseur de type 2, VMWare Player, pour la partie virtualisation. Cette association permet un effort d'installation et de configuration minimal tout en permettant de fournir un environnement d'exécution homogène même si les machines hôtes sont hétérogènes, grâce aux machines virtuelles. Ainsi l'utilisateur dépose son application et ses données dans la machine virtuelle et lance son exécution. Jusqu'au terme des calculs, l'utilisateur n'a à se soucier de rien, car la plateforme gère les pannes des machines hôtes en mettant en œuvre des mécanismes de tolérance aux pannes. La technique utilisée ici repose sur la méthode de sauvegarde/restauration. La seule contrainte imposée ici à l'utilisateur est d'indiquer dans son programme l'emplacement d'un point de sauvegarde.

Afin de présenter nos travaux, le reste de ce document est découpé en trois parties :

- la première partie définit le contexte scientifique, et s'articule autour deux chapitres :
  - le chapitre 1 présente les concepts du calcul scientifique, et plus précisément les méthodes numériques directes et itératives. Nous présentons également les deux grandes familles de parallélisation de ces méthodes, ainsi que les enjeux liés à la tolérance aux pannes. Enfin, la parallélisation des méthodes itératives est décrite plus en détails ;
  - le chapitre 2 décrit les architectures matérielles et les plateformes et environnements de calcul permettant la réalisation et l'exécution d'applications parallèles ;
- la seconde partie présente la problématique du placement des tâches d'applications itératives asynchrones sur des architectures distribuées, hétérogènes et volatiles. Cette partie est découpée en trois chapitres :
  - le chapitre 3 présente l'état de l'art concernant les algorithmes de placement. Nous y présentons notre problématique et nous donnons les définitions et modélisations utilisées ;
  - le chapitre 4 présente nos trois algorithmes de placement ainsi que leur intégration au sein de la plateforme JaceP2P-V2 ;
  - le chapitre 5 présente les expérimentations que nous avons menées afin d'évaluer les performances de nos algorithmes. Nous avons exécuté deux applications sur diverses architectures (degrés d'hétérogénéité et de volatilité différents). Les résultats obtenus montrent que nos algorithmes permettent de réduire significativement les temps d'exécution des applications et offrent des politiques de tolérance aux pannes efficaces. L'algorithme offrant les meilleures performances est MAHEVE.
- la troisième partie présente la mise en œuvre de notre plateforme de calcul virtualisée. Elle est composée de deux chapitres :
  - le chapitre 6 présente l'état de l'art et les motivations de l'utilisation des technologie de virtualisation pour l'exécution d'applications de calcul parallèles ;
  - le chapitre 7 décrit la mise en œuvre de notre plateforme de calcul utilisant des machines virtuelles pour l'exécution d'applications de calcul parallèles sur des architectures de type « enterprise desktop grid ». Nous présentons également des expérimentations montrant les performances de la plateforme. Nous avons exécuté deux applications aux caractéristiques différentes (l'une comportant des communications entre les tâches et l'autre non) sur une architecture de type « enterprise desktop grid ». Les résultats obtenus montrent que les performances de la plateforme sont bonnes et que le surcoût dû à son utilisation est acceptable.

Enfin nous concluons ce document en présentant les conclusions de nos travaux et leurs perspectives.

Première partie

Contexte scientifique

L'objectif de cette première partie est de définir le contexte scientifique nécessaire à la compréhension de cette thèse. Nous y donnons les notions essentielles et les définitions importantes qui sont utilisées dans les deux autres parties de ce document.

Dans le chapitre 1 nous présentons certains aspects du calcul scientifique. Dans un premier temps nous décrivons les méthodes de résolution de problèmes scientifiques. Nous y présentons les méthodes directes et les méthodes itératives. Dans un second temps nous présentons les différentes techniques de parallélisation de ces méthodes, à l'aide des deux principaux modèles les plus utilisés. Nous nous intéressons ensuite plus particulièrement aux méthodes itératives parallèles, en détaillant d'avantage le modèle itératif asynchrone. Enfin, nous définissons les enjeux et les techniques existantes pour la tolérance aux pannes des machines de calcul.

Le chapitre 2 présente dans un premier temps les plateformes d'exécution les plus répandues dans le domaine du calcul parallèle. Nous y décrivons les plateformes de calcul que sont les supercalculateurs, les clusters, les grilles de calcul, et les plateformes dites de calcul volontaire. Dans la seconde partie de ce chapitre nous détaillons quelques environnements d'exécution et de programmation permettant de concevoir et d'exécuter des applications parallèles sur des architectures distribuées. Nous présentons les principes de la norme MPI, puis nous détaillons les environnements ProActive, XtremWeb et XtremWeb-CH, ainsi que les environnements Jace, JaceP2P et JaceP2P-V2.

Chapitre 1

Calcul scientifique

Depuis très longtemps, les mathématiciens utilisent l'analyse numérique afin de résoudre des problèmes très difficiles et très longs. Ce domaine théorique est traité en partie à l'aide d'ordinateurs, en utilisant le calcul numérique – c'est le côté pratique de l'analyse numérique. Le calcul numérique est utilisé pour, par exemple, simuler des phénomènes naturels, tels que la résistance des matériaux, les prévisions météorologiques, mais aussi dans la médecine pour, par exemple, déterminer la localisation et les doses de radiations à donner à un patient pour traiter une tumeur. Tous ces traitements impliquent de gros volumes de données ainsi que de longs temps d'exécution.

Afin de réduire les temps d'exécution et obtenir la possibilité de traiter une plus grande quantité d'informations, les calculs sont répartis sur plusieurs machines en utilisant une décomposition des traitements et des données. Cette décomposition est le calcul parallèle, qui est décrit dans ce chapitre.

Tout d'abord, la section 1.1 présente les deux grandes familles de méthodes de résolution, que sont les méthodes directes et les méthodes itératives. Ensuite, la section 1.2 décrit les deux principales méthodes utilisées pour paralléliser une application. Puis en section 1.3 sont présentées plus en détails les versions parallèles des méthodes itératives. Enfin, la section 1.4 présente les enjeux de la tolérance aux pannes.

1.1 Méthodes numériques

Bon nombre de problèmes scientifiques peuvent être modélisés sous forme mathématique, à l'aide de systèmes d'équations, représentant les états du problème. Ces systèmes, linéaires ou non, peuvent être résolus à l'aide de méthodes numériques.

Considérons le système linéaire suivant :

$$Ax = b, x \in \mathbb{R}^n$$

dans lequel $A = (A_{i,j})_{1 \leq i,j \leq n}$ est une matrice carrée non singulière, et b est un vecteur de la forme $b = (b_1, b_2 \dots b_n)^T$. Il existe principalement deux classes d'algorithmes pouvant résoudre ce type d'équation : les *méthodes directes*, décrites en section 1.1.1, et les *méthodes itératives*, décrites en section 1.1.2.

1.1.1 Méthodes directes

Les algorithmes de cette première classe permettent de résoudre un problème en un nombre fini d'opérations élémentaires (si l'arithmétique utilisée est exacte, ce qui n'est pas le cas des machines actuelles). Lorsqu'on utilise un algorithme direct, il n'y a pas d'erreur de méthode, seules des erreurs d'arrondi apparaissent. À titre d'exemple, on peut citer les méthodes de *Cholesky* [40], *Gauss-Seidel* [69], *LU* [67], ou encore le *pivot de Gauss*. L'algorithme 1 décrit la résolution d'un système linéaire à l'aide de cette dernière.

Algorithme 1 : Méthode de résolution directe, algorithme de l'élimination de Gauss

Entrées : A (matrice), b (vecteur)

```
1  variables i, j, p
2  n ← taille(A)
3  pour i = 1 à n − 1 faire
4  |  p ← −1
5  |  choisir p, tel que i ≤ p ≤ n et A_pi ≠ 0
6  |  si p = −1 alors
7  |  |  pas de solution unique
8  |  |  retour nul
9  |  fin
10 |  si p ≠ i alors L_p ↔ L_i // échange des lignes
11 |  pour j = i + 1 à n faire
12 |  |  m_ij ← A_ij/A_ii
13 |  |  L_j ← L_j − m_ij × L_i
14 |  fin
15 fin
16 si A_nn = 0 alors
17 |  pas de solution unique
18 |  retour nul
19 fin
20 x_n ← b_n/A_nn
21 pour i = n − 1 à 1 faire
22 |  x_i ← [b_i − Σ_{j=i+1}^{n} A_ij × x_j]/A_ii
23 fin
```

L'algorithme 1 permet de résoudre un système linéaire de la forme $Ax = b$. Le but de l'algorithme est de transformer la matrice en une forme triangulaire supérieure, afin de résoudre plus facilement le système. Le principe est donc de mettre à zéro la partie inférieure de la matrice (sous la diagonale). Cette opération est réalisée par l'algorithme entre les lignes 3 et 15. Tout d'abord, on recherche le pivot sur la première ligne. Si celui-ci n'est pas trouvé, alors il n'existe pas de solution unique et l'algorithme ne pourra donner de solution.

Les méthodes directes permettent de résoudre des problèmes d'une certaine taille, mais s'avèrent trop coûteuses en temps sur de grands problèmes. De même, certains problèmes, comme par exemple le calcul des racines d'un polynôme, ne peuvent être résolus à l'aide de telles méthodes. Bon nombre de ces cas peuvent être résolus en utilisant des méthodes itératives, qui sont décrites dans la section suivante.

1.1.2 Méthodes itératives

Les méthodes itératives permettent elles aussi de résoudre, en autres, des problèmes linéaires, de la forme $Ax = b$, mais elles diffèrent des méthodes directes de par leur fonctionnement. En effet, les méthodes itératives procèdent par itérations successives d'un même bloc d'instructions, déterminant des solutions approximatives se rapprochant successivement de la solution cherchée, jusqu'à obtenir l'approximation voulue – on dit alors que l'algorithme a convergé vers la solution. Ces méthodes sont utilisées essentiellement lorsque les méthodes directes n'existent pas, ou lorsque le problème est mal conditionné ou comporte un trop grand nombre de variables. Pour ce dernier cas, les solutions successives limitent la propagation des erreurs. Les algorithmes de *Jacobi* [49], *GMRES* [60] (*Generalized Minimal Residual*) ou l'algorithme du *Gradient Conjugué* [57] appartiennent à cette classe.

Elles procèdent par approximations successives et construisent la solution à l'aide d'une séquence $x^{(k)}, k \in \mathbb{N}$ telle que

$$\lim_{k\to\infty} x^{(k)} = A^{-1}b$$

Le processus d'itération est arrêté lorsqu'une certaine précision, définie à l'avance, est atteinte.

De par leurs bonnes performances et leur stabilité, les algorithmes itératifs sont très utilisés dans de nombreuses applications scientifiques. À titre d'exemple, l'algorithme 2 présente la méthode de Jacobi, permettant de résoudre un système linéaire de manière itérative.

Algorithme 2 : Méthode de résolution itérative, algorithme de Jacobi

Entrées : A (matrice), b (vecteur)

```
variables i, j, k
n ← taille(A)
conv ← faux
k ← 0
choix d'un x^0
tant que ¬conv faire
    k ← k + 1
    pour i = 1 à n faire
        σ ← 0
        pour j = 1 à n faire
            si j ≠ i alors
                σ ← σ + A_ij × x_j^(k−1)
            fin
        fin
        x_i^(k) = (b_i − σ) / A_ii
    fin
    conv ← convergence ?
fin
```

Ici A représente la matrice, b le vecteur donné, x le vecteur solution, et *conv* un booléen indiquant si l'algorithme a convergé ou non. À la fin de l'exécution de l'algorithme, la solution du système se trouve dans le vecteur solution x. Le principe de l'algorithme est de construire une suite, dont le premier terme $x^{(0)}$ est donné, et la suite est de la forme $x^{(k+1)} = F(x^{(k)})$ avec $k \in \mathbb{N}$ et $F(x^{(k)})$ la fonction suite. Pour résoudre le système, on découpe la matrice en deux autres matrices, contenant pour l'une la diagonale, notée M, et l'autre le restant, notée $-N$. On obtient alors :

$$Ax = b \Leftrightarrow (M - N)x = b. \tag{1.1}$$

M est une matrice inversible. On obtient donc $Ax = b \Leftrightarrow Mx = Nx + b \Leftrightarrow x = M^{-1}Nx + M^{-1}b$. On peut donc écrire la suite sous la forme

$$x^{(k+1)} = F(x^{(k)}) = M^{-1}Nx^{(k)} + M^{-1}b, \tag{1.2}$$

avec $F(x)$ qui est une fonction affine. On retrouve dans l'algorithme la fonction de base sous la forme :

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j \neq i} A_{ij} x_j^{(k)} \right). \tag{1.3}$$

On remarque que l'algorithme a besoin des valeurs de l'itération k pour calculer les valeurs de l'itération $k+1$, ce qui implique d'avoir un stockage double du vecteur solution x.

Concernant l'arrêt de l'algorithme, il dépend de la convergence des résultats, dont la détermination est faite à la ligne 17. L'algorithme utilise la valeur du vecteur résidu $r^{(k)}$ et vérifie si :

$$\frac{||r^{(k)}||}{||b||} = \frac{||b - Ax^{(k)}||}{||b||} < \epsilon, \tag{1.4}$$

avec ϵ définissant la précision souhaitée.

Un point essentiel à prendre en compte ici est la convergence de l'algorithme. En effet, pour une méthode donnée, sa vitesse de convergence dépend des données initiales. Si les conditions de convergence ne sont pas « conformes », l'algorithme va alors diverger et par conséquent exécuter une boucle infinie. Par exemple pour la méthode de Jacobi, il faut que le rayon spectral de $M^{-1}N$ soit strictement inférieur à 1. Aussi, pour que l'algorithme converge pour n'importe quel $x^{(0)}$, il faut que la matrice A soit à diagonale strictement dominante.

Par conséquent, on peut noter ici que les algorithmes itératifs sont très sensibles aux données initiales, avec en plus des contraintes sur celles-ci. Ceci ne permet pas leur utilisation pour tous les problèmes. En effet, si les conditions de convergence ne sont pas réunies, l'algorithme ne pourra pas se terminer. Les principaux avantages de ces méthodes sont qu'elles permettent de traiter des problèmes bien plus grands que les méthodes directes, mais aussi leur parallélisation est plus simple et plus efficace. Dans la suite de ce document, nous nous intéressons à ce type de méthodes et nous présentons leur parallélisation en section 1.3.

1.2 Parallélisation

Comme énoncé précédemment, les calculs à effectuer portent sur des volumes de données très importants, et même en tenant compte de l'évolution des machines qui possèdent aujourd'hui plusieurs cœurs de calcul et de grandes quantités d'espace mémoire et de stockage, il est nécessaire, pour être efficace, de découper les calculs. Ceci est réalisé à l'aide du calcul parallèle, en découpant le calcul en tâches de calcul. Il est à noter que ce type de parallélisation n'est pas la seule possibilité, et qu'il existe d'autres techniques, comme par exemple la parallélisation des données ou des instructions (« pipelining » en anglais). La parallélisation en tâches est la méthode retenue pour les travaux présentés dans cette thèse.

Le principe du calcul parallèle par décomposition des données est de découper les données en plusieurs morceaux afin de distribuer la charge de calcul sur plusieurs machines. De ce fait, les différentes

machines vont travailler en parallèle sur différentes parties des données. Bien souvent, les calculs sur les données dites « frontières » (qui sont sur les bords du découpage) ont besoin des données se trouvant sur d'autres machines ; il faut alors effectuer des échanges de données entre les machines. Cet échange de données peut s'effectuer sous différentes formes, le plus souvent via l'envoi de messages contenant ces données, qui sont appelées « dépendances ».

Dans cette section sont tout d'abord présentés les deux principaux modèles de calcul parallèle. Le premier modèle est dit « sans communication », car les tâches n'ont pas de dépendance entre elles, et le second modèle est l'échange de messages.

1.2.1 Sans communication

Un des modèles les plus simples est le modèle sans communication – appelé « *fork-join* » en anglais. En effet, ce modèle repose sur le principe du découpage simple des traitements. Ici, chaque tâche est indépendante des autres ; chacune travaille sur ses propres données. Le travail le plus difficile concerne les phases de découpage et de rassemblement des données. La figure 1.1 présente le principe de tels algorithmes.

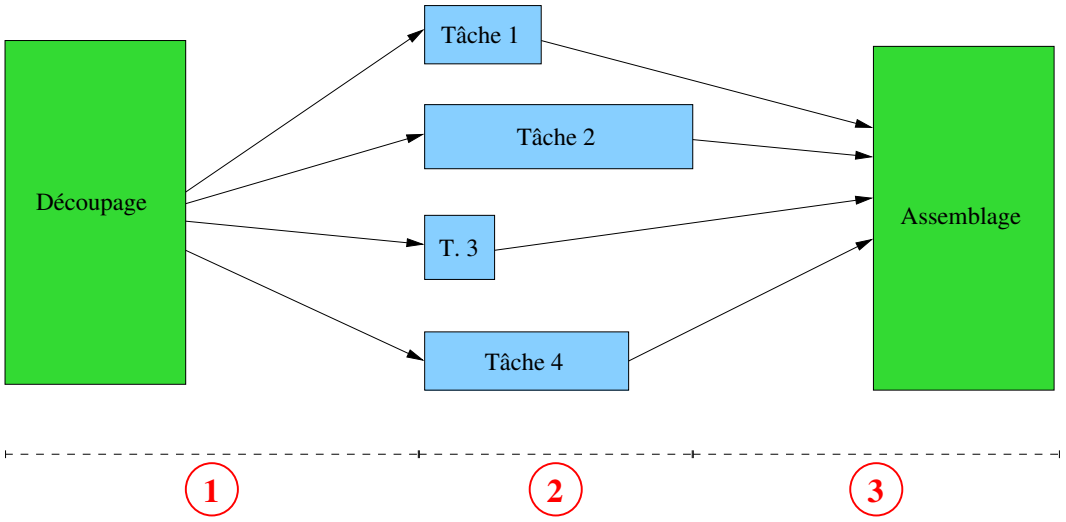

FIGURE 1.1 – Principe du découpage en tâches indépendantes

Comme le montre la figure 1.1, l'application se décompose en trois phases distinctes :

1. le découpage des données et traitements sous forme de tâches, et l'envoi de celles-ci vers chaque machine ;
2. le calcul à proprement parler, effectué par chaque machine avec des données locales ;
3. la réception et l'assemblage des résultats partiels.

La première phase est la plus délicate dans le sens où le découpage des données est primordial. En effet, il faut que chaque tâche puisse effectuer son traitement sans avoir recours aux données présentes sur d'autres machines. Un autre aspect important à prendre en compte dans cette première phase est l'équilibrage de charge. En effet, il se peut que les tâches ne représentent pas la même charge de calcul, mais aussi, les machines utilisées peuvent être de puissances différentes. Il risque donc d'y avoir un déséquilibre dans la phase de calcul – ainsi l'utilisation de la plateforme ne serait pas optimisée. La seconde phase, quant à elle, est simplement la phase de calcul. Le temps d'exécution des tâches peut varier, selon la répartition des données et des calculs, mais aussi suivant la puissance des machines

assignées à chaque tâche. Enfin, la troisième et dernière phase est celle de la réception des différents résultats et de leur exploitation.

L'application la plus connue utilisant cette forme de calcul parallèle est le projet Seti@Home [10]. Son principe est d'utiliser des machines volontaires pour traiter des parties de signaux provenant de l'espace, à la recherche de signaux démontrant la présence d'une vie extra-terrestre. Le projet récolte les signaux numériques provenant de l'écoute spatiale. Ces données sont fragmentées et distribuées à toute machine participant au projet (des centres de calcul, des ordinateurs de bureau, des ordinateurs personnels...). Les machines ne sont utilisées pour le projet que pendant leur phase d'inactivité, soit sous forme d'économiseur d'écran, soit sous forme de tâche de fond. Un fois le calcul effectué, le participant renvoie le résultat sur un serveur, qui le vérifie. Pour un sous-ensemble de données à traiter, celui-ci est envoyé à plusieurs participants. Ce mécanisme de sécurité permet de comparer les résultats et d'éviter que des machines se greffent au projet pour fausser les analyses. Au plus fort de son activité, le projet a pu réunir au même moment près de 500 TeraFlops de puissance de calcul.

Ce principe est aussi utilisé par d'autres projets du même type, comme par exemple Folding@Home [1] qui exécute des simulations sur les protéines. Nous pouvons également citer le cas des réseaux de neurones, où cette technique permet d'injecter en entrée de grands volumes d'information. Celles-ci seront traitées par apprentissage pour construire des réseaux de connaissance, qui seront ensuite utilisés pour trouver des solutions plus rapidement que des méthodes plus classiques.

1.2.2 Passage de messages

Un autre modèle très utilisé est l'échange de données de dépendances par l'envoi et la réception de messages, appelé « échange par passage de messages » (« message passing » en anglais).

Un des modèles les plus utilisés est MPI, pour « Message Passing Interface », qui est décrit plus en détails en section 2.2.1. Le principe est assez simple : à la fin d'une phase de calcul, chaque tâche envoie un message à ses voisins de calcul, dont les calculs dépendent des résultats disponibles, contenant les données nécessaires à la suite de l'exécution. Généralement, un message contient plusieurs informations :

- l'identifiant de l'expéditeur ;
- l'identifiant du destinataire ;
- l'identifiant de séquence du message (permettant de pallier les temps de latence entre plusieurs envois et permet de détecter la perte de messages) ;
- les données (ou dépendances) ;
- éventuellement la position des données dans la structure de réception.

Suivant la mise en œuvre du système de gestion des messages, aussi bien au niveau de l'envoi que de la réception, et de la conception de l'algorithme, toutes ces données ne sont pas forcément utilisées.

Pour les mécanismes d'envoi et de réception, plusieurs solutions existent. Dans certains cas, les deux sont bloquants, c'est-à-dire que l'algorithme attend que le message envoyé soit reçu, et pour la réception, il reste bloqué tant qu'il n'a pas reçu de données. Généralement, l'utilisation des envois et réceptions bloquants résultent en phases de synchronisation des tâches – assurant la cohérence des données et le non blocage de l'application. Pour d'autres algorithmes, les mécanismes sont non bloquants, c'est-à-dire que les messages sont envoyés sans attendre qu'ils soient arrivés à destination, et de même pour la réception, si un message est arrivé il est pris en compte, sinon l'algorithme continue sans ces données.

Chaque programmeur utilise la méthode qui lui convient et qui est également en phase avec l'algorithme, qui peut supporter ou non le choix d'échanges bloquants ou non – il est souvent plus délicat d'utiliser les versions non bloquantes que bloquantes. Le plus souvent, le modèle retenu est l'envoi non bloquant avec réception bloquante.

1.3 Méthodes itératives parallèles

Il existe deux manières d'envisager la parallélisation des algorithmes itératifs : les méthodes synchrones et les méthodes asynchrones. Les deux méthodes sont assez différentes et dépendent fortement du contexte dans lequel se déroulent les communications.

1.3.1 Méthodes itératives synchrones

Les algorithmes les plus classiques, que ce soit pour la programmation standard ou plus orientée haute performance, sont des algorithmes dits *synchrones*. On distingue ici 2 phases, qui dans certains cas ont la même importance : la phase de calcul qui se sert de la puissance du processeur local, et la phase de communication qui n'occupe que la liaison réseau avec les autres machines.

Pour qualifier un algorithme de synchrone, les deux phases ne sont pas obligatoirement synchrones toutes les deux ; généralement, la phase de calcul est toujours synchrone, mais la phase de communication peut être asynchrone. On a alors un type d'algorithme dit ISCS, pour « Itérations Synchrones, Communications Synchrones », illustré par la figure 1.2 (ici les envois et les réceptions sont bloquants) ; et un autre type dit ISCA, pour « Itérations Synchrones, Communications Synchrones », illustré par la figure 1.3 (ici les envois sont non bloquants et les réceptions sont bloquantes).

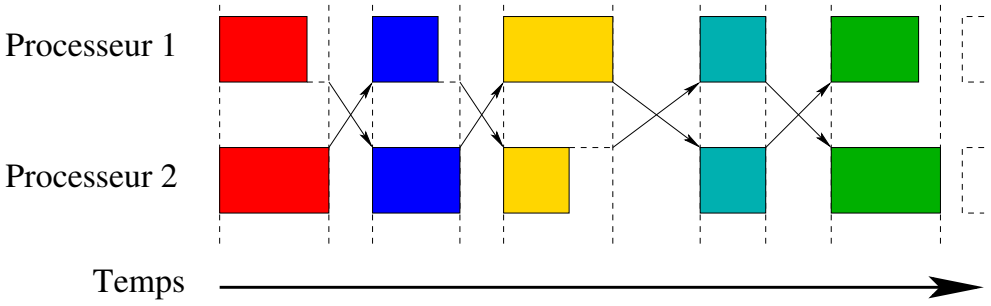

FIGURE 1.2 – Deux processeurs en calcul itératif synchrone (ISCS)

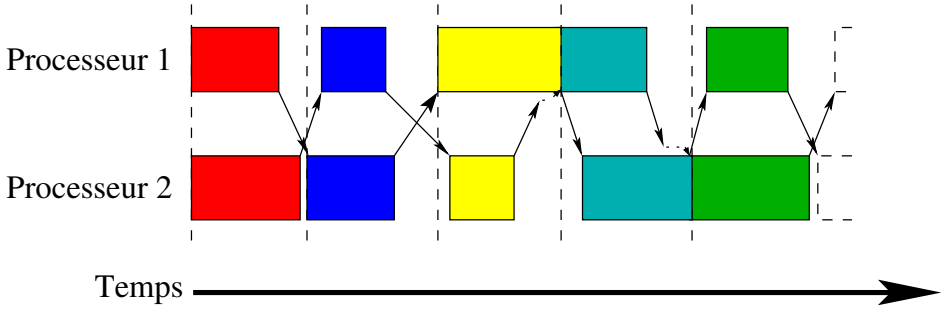

FIGURE 1.3 – Deux processeurs en calcul itératif synchrone avec communications asynchrones (ISCA)

Pour ces algorithmes, le nombre d'itérations est égal au nombre d'itérations en mode séquentiel – c'est-à-dire qu'à chaque exécution on obtient toujours le même nombre d'itérations. On peut remarquer qu'à chaque instant tous les processeurs calculent la même itération. On remarque également qu'il existe de nombreux temps d'attente sur chacun des processeurs (représentés sur la figure par les espaces blancs entre les tâches). Ces temps d'attente correspondent aux synchronisations entre les itérations et/ou les temps de communication. Ils peuvent dégrader considérablement les performances

de l'algorithme parallèle. On remarque enfin que les algorithmes ISCA permettent un recouvrement partiel des communications par des calculs.

1.3.2 Méthodes itératives asynchrones

Les algorithmes itératifs asynchrones [13] permettent de s'affranchir de l'attente des autres processeurs. En effet, le principe est que non seulement les communications sont asynchrones (envois et réceptions non bloquants), mais aussi les phases de calcul. Pour ce faire, des algorithmes spécifiques sont mis en place avec pour principe de ne traiter que le dernier message reçu, comme étant celui comportant les données les plus à jour – contrairement au mode synchrone où sont traitées toutes les données dans l'ordre où elles sont arrivées. Si jamais il n'y a pas de nouvelles données en attente, le programme continue alors son exécution avec les données en sa possession – en l'occurrence les dernières qu'il vient de calculer. Ces algorithmes sont appelés IACA, pour « Itérations Asynchrones, Communications Asynchrones ».

Le schéma d'exécution permet, comme on peut le voir sur la figure 1.4, de ne pas avoir de temps morts, optimisant ainsi au mieux l'utilisation des processeurs.

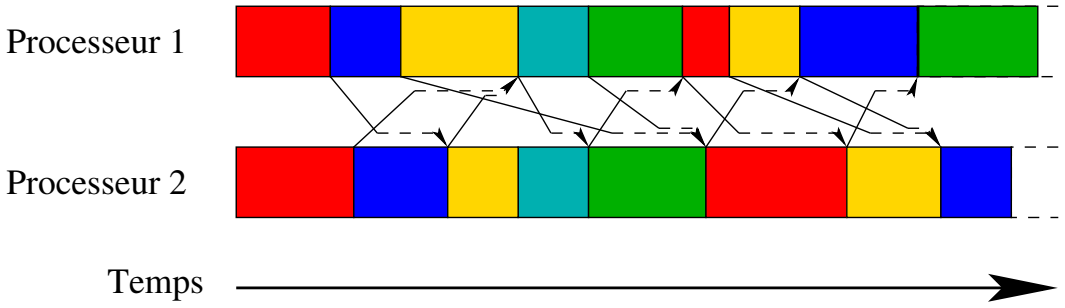

FIGURE 1.4 – Deux processeurs en calcul itératif asynchrone (IACA)

Principes algorithmiques

Prenons ici l'exemple de la résolution d'un système linéaire de la forme $Ax = b$. Le vecteur solution x contenant toutes les inconnues est découpé en m blocs. Dans le modèle asynchrone, contrairement au modèle synchrone, toutes les m inconnues ne sont pas mises à jour à chaque itération. Ceci provient du fait que les communications ainsi que les itérations sont asynchrones. Les principes de ce modèle sont :

- les m blocs d'inconnues répartis entre les m tâches peuvent être mis à jour dans n'importe quel ordre, et il est possible que certains blocs ne soient pas mis à jour régulièrement. Cependant, tous les blocs évoluent à un moment ou un autre, aucun reste dans son état initial ;
- à un temps t, chaque tâche vérifie si elle a reçu des messages de ses dépendances. Dans l'affirmative, elle met à jour les inconnues en utilisant la dernière version disponible. En effet, dans ce modèle, il est possible que les dépendances d'une tâche aient effectué plusieurs itérations avant qu'elle n'en finisse une, et lui aient envoyé plusieurs fois des mises à jour. Le principe ici est de ne tenir compte que de la dernière mise à jour de chaque message, afin de progresser plus rapidement. Dans le cas contraire, la tâche continue son calcul sans faire de mise à jour.

Ainsi les tâches n'attendent pas de recevoir des messages de leurs dépendances pour continuer leur calcul. Ceci permet à des tâches, à un même instant t, d'être dans des itérations différentes – suivant la puissance de calcul de leur machine respective.

Avantages et inconvénients

Avantages Ces algorithmes ne sont pas simples à mettre en œuvre, mais offrent quelques avantages non négligeables :

- pas de temps mort : comme le montre la figure 1.4, après la fin de chaque itération, chaque tâche envoie ses données de dépendances à ses voisins, et n'attend pas la réception des messages de ceux-ci pour commencer l'itération suivante. Il n'y a donc aucun temps mort entre deux itérations. Ici, les communications sont totalement recouvertes par du calcul ;
- pas de synchronisation : comme les envois et les réceptions de messages sont totalement asynchrones et que les tâches n'attendent pas de données pour commencer une nouvelle itération, alors plus aucune synchronisation n'est nécessaire. Ceci permet aux machines les plus rapides d'avancer très vite dans le calcul, sans être ralenties par les plus lentes. Cette caractéristique permet au modèle asynchrone d'être moins sensible à l'hétérogénéité aussi bien des machines que des liens de communication lors d'exécutions sur des plateformes distribuées ;
- tolérance aux pertes de messages : en déduction des deux points précédents découle un avantage majeur de ce modèle, qui est la tolérance aux pertes de messages. En effet, comme décrit précédemment, une tâche n'attend pas la réception de données pour commencer une nouvelle itération, en utilisant simplement les dernières données arrivées. Ceci permet de rendre totalement transparente la perte de messages – il faut cependant que des données arrivent de temps en temps pour permettre à l'algorithme de progresser. Par extension, la panne transitoire d'une machine n'affecte en rien l'exécution des autres tâches. Cette caractéristique est très importante dans des environnements volatils.

Pour mieux illustrer ce dernier avantage, les figures 1.5 et 1.6 montrent les effets de la panne d'une machine lors d'un calcul, pour respectivement le modèle synchrone et le modèle asynchrone.

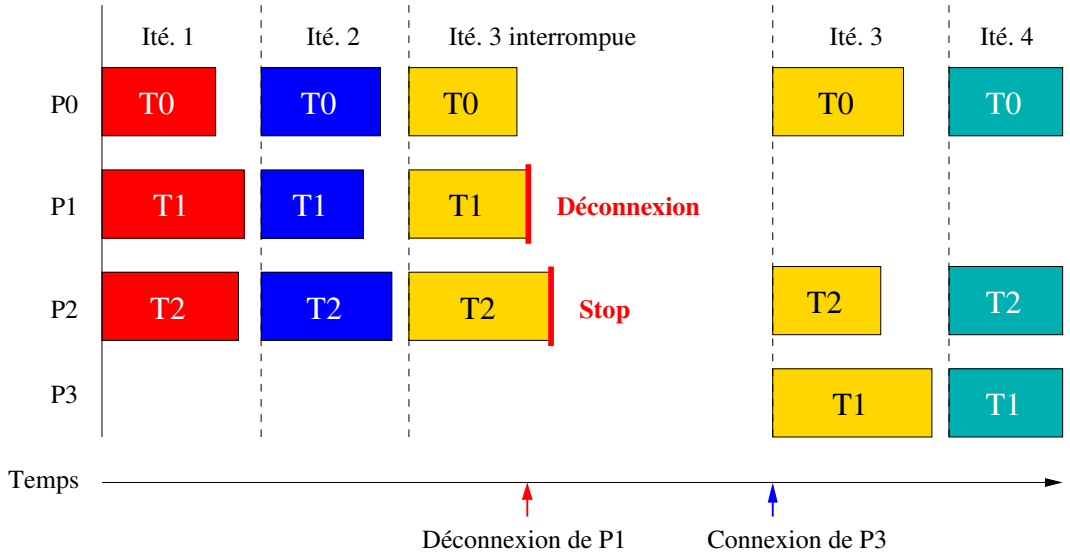

FIGURE 1.5 – Exécution d'un algorithme itératif synchrone dans un environnement volatil

L'application exécutée comporte trois tâches interdépendantes. Tout d'abord, concernant la version synchrone, à la fin de chaque itération une sauvegarde est effectuée et est envoyée vers un serveur de stockage. On remarque que toutes les tâches effectuent une synchronisation, résultant en une perte de

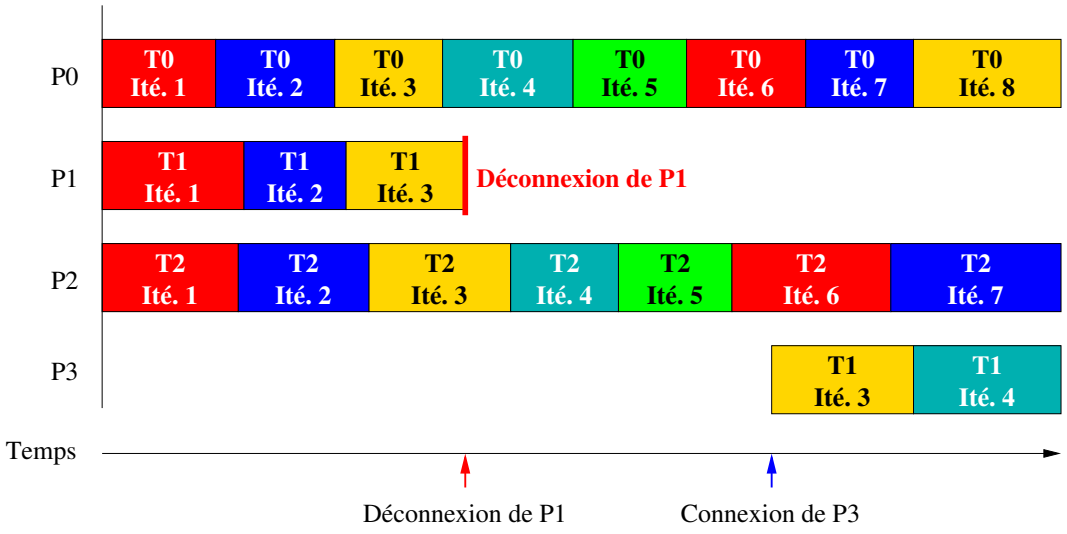

FIGURE 1.6 – Exécution d'un algorithme itératif asynchrone dans un environnement volatil

temps de calcul. Ensuite, lorsqu'à l'itération 3 la machine P1 exécutant la tâche T1 se déconnecte, la tâche T2 est arrêtée après la détection de la panne, alors que la tâche T0 est terminée. L'application reste ainsi bloquée tant qu'aucune machine disponible ne rejoint la plateforme. Lorsque la machine P3 se connecte, le calcul peut reprendre – P3 exécute alors la tâche T1. Afin de relancer les calculs, les sauvegardes des trois tâches, effectuées à la fin de l'itération 2, sont déployées et l'exécution de l'application peut reprendre. Il est à noter ici que la tâche T0 calcule de nouveau l'itération 3.

Concernant cette fois le modèle asynchrone, qui exécute la même application, les sauvegardes sont effectuées à la fin de chaque itération, suivant le modèle des sauvegardes non coordonnées, décrites en section 1.4.4. Il est à noter que ce mécanisme de sauvegarde n'induit aucune synchronisation, donc aucun temps mort entre les itérations. Les sauvegardes sont réparties sur les machines de calcul et non plus stockées sur un serveur dédié. La grande différence avec le modèle synchrone est visible lorsque P1 tombe en panne : les tâches T0 et T2 continuent leur exécution normalement. Ainsi, lorsque la machine P3 se connecte à la plateforme, elle récupère la dernière sauvegarde de la tâche T1, effectuée à la fin de l'itération 2. Ainsi, seule la tâche T1 calcule de nouveau l'itération 3, pendant que la tâche T0 calcule sa septième itération et la tâche T2 sa sixième itération.

Ceci démontre bien les avantages du modèle itératif asynchrone lorsque la plateforme d'exécution utilisée comporte des éléments volatils et hétérogènes. Les expérimentations décrites dans [12] montrent que ces algorithmes permettent d'obtenir de bonnes performances lorsqu'ils sont exécutés sur des architectures distribuées avec de fortes latences entre les machines de calcul.

Inconvénients Bien que le modèle itératif asynchrone présente des avantages non négligeables, notamment lorsque les exécutions se déroulent dans des environnements volatils et hétérogènes, il existe des inconvénients :

- la compatibilité : malheureusement, ce modèle ne peut être appliqué à tous les algorithmes itératifs, comme par exemple GMRES ou le calcul du gradient conjugué. En effet, ces algorithmes ont besoin de recevoir à la fin de chaque itération des mises à jour de données, ceci afin d'assurer leur convergence, en un nombre fini d'itérations. Dans le cas contraire, ces algorithmes ne convergent pas ;
- le nombre d'itérations : de par sa nature, ce modèle effectue plus d'itérations que le modèle synchrone. Ainsi lorsque la plateforme d'exécution est homogène et que les liens de communication

sont très performants, les performances du modèle IACA sont généralement inférieures à celles du modèle synchrone [13] ;
- la détection de convergence : comme les tâches ne reçoivent pas systématiquement de mise à jour de leurs dépendances à la fin de chaque itération, les détections classiques de convergence ne fonctionnent pas. En effet, si durant plusieurs itérations des inconnues ne sont pas mises à jour, alors le résidu se trouvera en-dessous du seuil indiqué, entraînant alors une fausse détection de convergence. Des mécanismes dédiés existent afin de pallier ce problème. Le principe général est de lancer un algorithme de consensus quand une convergence locale est détectée. Le problème devient un peu plus complexe si l'on considère des exécutions dans des environnements volatils. Plus de détails sur un tel algorithme peuvent être consultés dans [23].

Ces algorithmes sont performants dans les cas où les données à traiter sont de taille assez conséquente et que les machines sont hétérogènes avec des liaisons réseaux tout autant hétérogènes. Ceci permet de pallier les problèmes dûs à l'environnement matériel. Les temps de communication sont recouverts par des temps de calculs. Ainsi le modèle IACA semble particulièrement bien adapté à des architectures distribuées et hétérogènes, comme les grilles de calcul.

1.4 Tolérance aux pannes

Un point essentiel à prendre en compte lors de l'exécution d'une application sur une machine et a fortiori sur plusieurs, est le fait que la machine peut, pour diverses raisons, « tomber en panne ». Il peut arriver à une machine d'avoir une défaillance matérielle et/ou logicielle, et pour le cas où plusieurs machines sont utilisées en réseau, une panne des liens de communication peut survenir. Pour pallier ces pertes de calculs, et donc de résultats et de temps, il existe des mécanismes dits de tolérance aux pannes. Les deux principaux mécanismes utilisés sont la réplication des tâches et la sauvegarde/restauration des calculs. Cependant, pour que ces méthodes soient utilisables, il faut pouvoir détecter efficacement les pannes qui surviennent.

1.4.1 Détection de panne

La détection des pannes dans un mécanisme de tolérance aux pannes est un point essentiel. En effet, sans une bonne détection des pannes, beaucoup de temps de calcul peut être perdu. C'est pourquoi il faut qu'elle soit la plus rapide possible, afin de déclencher les mécanismes de restauration. De plus, le mécanisme de détection doit être le plus transparent possible pour l'application, c'est-à-dire ne pas surcharger ni le réseau ni les machines, aussi bien celles effectuant des calculs que celles assurant la gestion de la plateforme. Généralement, deux familles de détection sont utilisées : la détection standard et la détection utilisant un « protocole de bavardage ».

Détection standard

Cette famille de mécanismes repose sur l'envoi et/ou la réception de messages indiquant si une machine n'est pas en panne [31]. Ici, on se place du point de vue d'une machine de calcul ; c'est-à-dire qu'un envoi est un message partant de la machine de calcul vers l'entité de gestion, et une réception est l'inverse. On peut distinguer ici trois modèles :

- envoi de messages : le principe de ce modèle est que l'entité de calcul envoie périodiquement un message à une unité de gestion, lui signifiant qu'elle est toujours en vie. Ce message est appelé un battement de cœur, ou « heartbeat » en anglais. Après un certain laps de temps sans recevoir de

heartbeat d'une machine, l'unité de gestion la déclare alors « en panne » et peut par conséquent déclencher les mécanismes de restauration. Ce modèle devient vite pénalisant et inefficace dès lors que le nombre de machines devient trop important pour une seule unité de gestion. Un autre inconvénient peut provenir de la trop forte latence du réseau, empêchant alors les messages d'arriver à temps, entraînant une fausse détection de panne ;
- réception de messages : le principe de ce modèle est que cette fois, c'est l'unité de gestion qui envoie un message de heartbeat à l'entité de calcul, qui doit lui répondre. De la même façon que précédemment, si cette entité ne répond pas dans un certain laps de temps, l'unité de gestion la considère alors comme en panne. Ce modèle est plus efficace dans la détection que le premier modèle. Cependant, l'inconvénient majeur est que les messages de heartbeat sont doublés, de par le fait que l'unité de gestion initie une séquence à laquelle l'entité de calcul doit répondre ;
- envoi/réception de messages : ce troisième modèle tire partie des points forts des deux précédents. En effet, afin d'effectuer la détection de pannes, deux phases sont nécessaires. La première se réferre au premier modèle, en laissant l'initiative à l'entité de calcul d'envoyer les messages de heartbeat. Passé un certain délai durant lequel celle-ci n'a pas envoyé de messages, c'est alors l'unité de gestion qui prend l'initiative et envoie une requête à l'entité de calcul mise en cause. Si celle-ci ne répond pas dans un certain délai, elle est alors déclarée comme en panne. Dans le cas contraire, le système lui redonne l'initiative. Ce modèle permet d'obtenir une meilleure détection des pannes, qui est plus sûre, mais qui peut prendre un peu plus de temps.

Ces mécanismes de détection de pannes sont efficaces et simples à mettre en œuvre au sein d'une plateforme de calcul. Cependant, toutes ces méthodes possèdent un inconvénient majeur qui est le non passage à l'échelle. En effet, comme toute la gestion est centralisée au niveau de l'unité de gestion, celle-ci ne peut gérer de manière efficace des centaines voire des milliers de requêtes. De plus, lorsque beaucoup de machines sont surveillées, si plusieurs tombent en panne en même temps, les temps de réaction de l'entité de gestion peuvent être très longs. Ainsi, ces mécanismes peuvent être utilisés pour de petites architectures, mais ils doivent être remplacés par d'autres plus performants pour les architectures comportant beaucoup de machines. Des approches existent pour pallier ces inconvénients, comme par exemple la mise en place de plusieurs unités de gestion, chacune surveillant un groupe de machines. Cependant, de par leur principe, si dans ces mécanismes une ou plusieurs unités de gestion tombent en panne, alors toute la politique de tolérance aux pannes devient invalide.

Protocole de bavardage

Un autre modèle de détection de pannes est le protocole de bavardage [59], « gossip protocol » en anglais. Cette fois, le mécanisme est totalement décentralisé. La détection est effectuée par les entités de calcul elles-mêmes. L'unité de gestion est ainsi délester de la charge de détection, et intervient uniquement pour déclencher les mécanismes de restauration lorsqu'elle est informée d'une panne.

Le principe de ce modèle est que chaque entité de calcul possède une liste des autres entités. Pour chaque entrée de cette liste représentant une entité de calcul, on y retrouve son adresse ainsi qu'un compteur représentant son heartbeat. À chaque pas de temps défini, chaque entité incrémente son propre compteur et envoie la liste mise à jour à une voisine sélectionnée au hasard. S'ensuit alors que si après un certain laps de temps, une entité A s'aperçoit que l'entité B n'a toujours pas incrémenté son compteur, alors A suspecte B d'être tombée en panne. Cependant, afin de s'en assurer, une phase de détection en consensus est déclenchée. Dans l'affirmative, A avertit l'unité de gestion de la panne de B, et les mesures de restauration sont exécutées. L'inconvénient majeur de cette méthode est qu'il est possible qu'une entité de calcul ne reçoivent jamais de message de mise à jour, et donc déclare les autres entités comme étant tombées en panne.

Pour pallier cet inconvénient, l'utilisateur peut augmenter le délai de détection, mais ceci a l'incon-

vénient de ralentir la détection des vraies pannes. Une autre approche consiste à utiliser une meilleure distribution des messages. En effet, il suffit d'utiliser par exemple un algorithme choisissant chaque entité de calcul de la liste « à tour de rôle ». À chaque tour d'envoi, l'entité destinataire change, tout en assurant le fait que toutes les entités vont recevoir un message. Ceci permet d'assurer que dans un délai borné chaque entité recevra un message de mise à jour, empêchant ainsi de fausses détection de pannes. De plus, ce mécanisme permet de réduire le délai de détection. D'autres améliorations sont proposées dans [35].

1.4.2 Réplication de tâches

Un mécanisme en l'apparence assez simple et intuitif est de répliquer les tâches de calcul sur plusieurs machines. Ainsi, si une machine devient indisponible, le « clone » de son calcul continue sur une autre machine.

La méthode la plus couramment utilisée est la « réplication active ». Comme son nom l'indique, plusieurs instances du même calcul s'exécutent en même temps. Cette méthode a l'avantage de masquer de manière quasi parfaite les défaillances de machines. Cependant, ceci oblige le programmeur à faire attention à bien gérer les différentes instances de calcul. En effet, si plusieurs machines effectuent le même calcul mais à des vitesses différentes (machines hétérogènes), il faut gérer le décalage, notamment lors des phases d'échanges de messages par exemple. Un autre inconvénient est la consommation supplémentaire de ressources. En règle générale, les clones de calcul sont déployés sur d'autres machines, qui n'auraient pas participé au calcul sans ce mécanisme de tolérance aux pannes. Le cas le plus défavorable est lorsque qu'il n'y a pas assez de machines libres pour cloner les calculs. La situation est telle, que sur une même machine s'exécutent alors une tâche de calcul et un ou plusieurs clones d'autres tâches. De plus, la duplication des échanges de messages surcharge le réseau. Ceci résulte alors en une perte de performances pour l'application toute entière.

1.4.3 Sauvegarde des messages

Ce modèle repose sur l'archivage des messages reçus [9]. En fait, pour chaque message de mise à jour des données envoyé, une copie est sauvegardée sur un serveur dédié. Ainsi, lorsqu'une machine tombe en panne, celle la remplaçant a simplement besoin de l'image initiale de la tâche et des archives des messages la concernant. Ensuite, tous les messages sont appliqués sur l'image initiale afin de retrouver le dernier état connu pour la tâche. Cette méthode a le principal inconvénient de rapidement saturer le serveur de sauvegarde, surtout lorsque le nombre de tâches est très grand.

Pour pallier ce problème, il est possible de mettre en place une politique de sauvegarde distribuée. En effet, il suffit qu'en lieu et place de l'envoi des sauvegardes sur un serveur, l'entité de calcul conserve une copie locale et envoie des copies à d'autres entités. Ainsi, seulement lorsque les sauvegardes deviennent trop grandes, celles-ci sont envoyées à un serveur sûr. Lors d'une panne, la machine remplaçante peut récupérer les données sur d'autres machines. Un inconvénient survient lorsque dans cette phase de « remise à niveau », la nouvelle machine ne peut pas avoir accès à certaines données. Dans ce cas, toutes les autres tâches doivent reprendre leur sauvegarde d'origine et rejouer tous les messages afin de se mettre au même niveau que celle n'ayant pas accès à toutes les données. Ceci résulte en un mécanisme long et fastidieux. De plus, dans des environnements volatils, où les machines peuvent apparaître et disparaître rapidement, un tel mécanisme n'est pas envisageable.

1.4.4 Sauvegarde / redémarrage

Un autre mécanisme permettant la tolérance aux pannes est la méthode de sauvegarde/redémarrage [56], « checkpoint/restart » en anglais. Cette méthode est un peu plus complexe à mettre en œuvre que la réplication.

Le principe ici est de sauvegarder l'état courant du calcul (uniquement les données essentielles) et d'envoyer cette sauvegarde sur une ou plusieurs autres machines (suivant le degré de tolérance aux pannes souhaité). Ici, deux sortes de mécanismes existent :

- les sauvegardes coordonnées : cette méthode, appelée « coordinated checkpointing [22] » en anglais, requiert une synchronisation de toutes les tâches. Une fois toutes les tâches synchronisées, une sauvegarde est créée. Avec cette méthode, si une machine tombe en panne, toutes les autres machines participant au calcul restent bloquées jusqu'à que la remplaçante soit mise en place. Une fois la sauvegarde restaurée, toutes les autres tâches doivent recommencer à partir du même « numéro » de sauvegarde, et continuer ainsi l'exécution de l'application ;
- les sauvegardes non coordonnées : dans cette méthode, appelée « uncoordinated checkpointing » [28] en anglais, il n'y a plus de synchronisation. En effet, lorsque les conditions sont réunies, une tâche effectue sa sauvegarde et procède à son transfert. Lors d'une panne, la machine remplaçante récupère la dernière sauvegarde de la tâche, alors que les autres continuent leur exécution, comme si de rien n'était. Il est à noter que ce mécanisme n'est pas supporté par toutes les applications – par exemple ce modèle convient très bien pour les applications reposant sur le modèle itératif asynchrone.

Bien souvent, les sauvegardes sont envoyées sur un serveur de stockage. Ceci entraîne alors un goulot d'étranglement lors des phases de sauvegarde, surtout lors de l'emploi de la première méthode. De plus, un tel mécanisme ne permet pas le passage à l'échelle et n'est pas envisageable dans des environnements volatils. Pour pallier ce problème, une solution consiste à distribuer les sauvegardes sur les voisins de calcul (le nombre peut être donné comme paramètre par l'utilisateur). Bien que performante, cette solution ne peut pas prévenir l'application contre tous types de pannes.

Un autre inconvénient de ce mécanisme est le fait que le programmeur doit choisir quels sont les données et résultats à sauvegarder, comment effectuer la sauvegarde, mais également comment redémarrer le calcul à partir des données contenues dans la sauvegarde. Un autre inconvénient peut provenir du temps de redémarrage d'une tâche de calcul, qui comprend le temps de transfert et le repositionnement dans le calcul, qui est fonction de la complexité et de la taille des données sauvegardées. L'avantage de cette méthode est qu'elle ne consomme que peu de ressources supplémentaires, car les sauvegardes sont effectuées sur les machines participant au calcul. Ainsi seul de l'espace disque sera consommé et un peu de bande passante pendant les phases de transfert.

1.5 Conclusion

Dans ce chapitre ont été présentées les différentes formes de calcul numérique parallèle parmi les plus utilisées. Chaque type de calcul doit être choisi, dans la mesure du possible, en fonction de l'architecture sur laquelle va s'exécuter l'application. Aussi, en fonction de l'algorithme de calcul en lui-même, les possibilités de parallélisation ne sont pas les mêmes.

Ainsi, le modèle sans communication est utilisé dans les cas où le découpage des données est facile et que les calculs restent locaux aux tâches, c'est-à-dire qu'elles n'ont pas besoin de données externes. Les modèles avec communications, reposant sur des méthodes directes ou itératives, nécessitent un travail plus long dans l'élaboration de la parallélisation des applications. En effet, une grande attention doit être portée au découpage des données, mais également aux dépendances entre les tâches de calcul,

afin de les ajuster au mieux. Les méthodes directes sont plus difficiles à paralléliser que les méthodes itératives de par leur nature. De plus, lorsque que le volume de données à traiter est très important ou lorsque le nombre d'inconnues est très grand, les méthodes directes peuvent s'avérer inefficaces, de par leur temps d'exécution et les risques importants de propagation de l'erreur.

Concernant les méthodes itératives, plus simples à paralléliser que les méthodes directes, elles permettent de traiter des problèmes de très grande taille et/ou avec un nombre très grand d'inconnues, tout en limitant la propagation de l'erreur. Le modèle itératif synchrone, qui est le plus utilisé, souffre de pertes de performances dûes aux nombreuses synchronisations qu'il nécessite, notamment dans des environnements hétérogènes (aussi bien au niveau des machines que des liens de communication). Le modèle itératif asynchrone quant à lui permet de pallier ces problèmes, au prix d'un nombre d'itérations plus élevé. Il est important de noter que tous les algorithmes itératifs ne peuvent pas être transformés suivant le modèle asynchrone, qui impose de fortes conditions sur l'algorithme et les données.

Un autre point important à prendre en compte lors de la parallélisation d'un algorithme en vue de son exécution sur une architecture distribuée (mais aussi dans le cas d'une exécution sur une seule machine) est la mise en place de mécanismes de tolérance aux pannes. La méthode est à choisir en fonction de la facilité d'intégration dans l'algorithme, et suivant l'architecture d'exécution cible. Certaines méthodes sont plus adaptées à certains algorithmes et à des architectures particulières.

Chapitre 2

Plateformes et environnements d'exécution

Après avoir décrit les méthodes et algorithmes parallèles dans le chapitre précédent, celui-ci est consacré aux plateformes et aux environnements d'exécution d'applications parallèles. Les choix effectués par le programmeur et l'utilisateur dépendent de leurs préférences, mais aussi répondent à certaines contraintes, suivant la nature de l'algorithme et/ou les moyens matériels mis à leur disposition.

Dans un premier temps, nous nous intéressons aux plateformes d'exécution, depuis les supercalculateurs, en passant par les clusters et les grilles, jusqu'aux plateformes de calcul volontaire. Chacune de ces plateformes correspond à des besoins et des moyens particuliers. Dans un second temps, seront présentés les principaux environnements d'exécution, notamment ceux utilisés pour la réalisation des travaux présentés dans cette thèse.

2.1 Plateformes d'exécution

L'objectif de cette section est de présenter les différentes plateformes d'exécution utilisées dans le calcul numérique parallèle. Tout d'abord sont présentés les supercalculateurs, qui connaissent aujourd'hui un regain d'intérêt, puis les clusters, les grilles, et enfin les plateformes dites de calcul volontaire.

2.1.1 Supercalculateurs

Les supercalculateurs sont la première architecture utilisée pour le calcul numérique parallèle. En effet, dès les années 1960, est apparu le premier vrai supercalculateur, construit par la société Cray. À l'époque, sa puissance de traitement était estimée à 3MégaFlops (soit 3×10^6 opérations par seconde). Au cours des années 1990, les supercalculateurs ont perdu l'engouement qu'ils ont suscité pendant de nombreuses années – seuls quelques constructeurs, tels que IBM, Cray et Bull par exemple, ont su garder leur savoir faire. La principale raison de cette baisse d'intérêt est leur coût, aussi bien d'acquisition que de maintenance ; surtout à une époque où apparaissent les clusters (qui sont décrits dans la section suivante).

Le principe d'un supercalculateur est de rassembler les composants de plusieurs ordinateurs en une seule machine. Ceci permet d'avoir beaucoup de mémoire et d'espace disque, mais aussi de nombreux processeurs, tous reliés entre eux par des liens de communication très performants. La figure 2.1 présente le supercalculateur Cray-2.

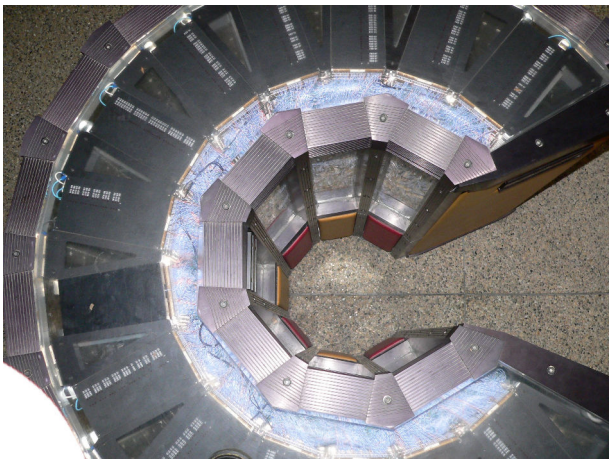

FIGURE 2.1 – Le supercalculateur Cray-2 (vu du dessus)

Aujourd'hui, les supercalculateurs reviennent sur le devant de la scène, dans ce qu'on appelle l'« exascale [1] ». Leur construction a cependant bien évolué. En effet, à leur début, les supercalculateurs représentaient des machines d'un bloc composées de processeurs spécialisés ; aujourd'hui ils ressemblent d'avantage à des « clusters ». Les raisons sont un coût de fabrication moins élevé, une plus grande flexibilité de construction et surtout une maintenance plus aisée – il est plus facile de changer un composant dans ce nouveau genre de machine.

Il existe depuis un classement mondial de ces nouveaux supercalculateurs, le Top500 ; preuve du renouveau d'intérêt pour cette architecture. Actuellement, le supercalculateur le plus performant est le « K Computer » du constructeur Fujitsu, pour l'université de RIKEN à Kobe au Japon, possédant une puissance de 8PétaFlops (soit 8×10^{15} opérations par seconde). Pour atteindre cette puissance, pas moins de 68544 processeurs sont utilisés, possédant chacun 8 cœurs cadencés à 2GHz, représentant donc un total de 548352 cœurs de calcul. Les processeurs sont des Sparc64 VIIIfx, chacun accompagné de 16Go de mémoire RAM, soit un total d'environ 1Po (1 million de giga) de mémoire. Il est à noter que le projet initial ne sera pas terminé avant novembre 2012 avec près du tiers de puissance en plus. Avec des machines aussi puissantes et complexes, de nouveaux axes de recherche apparaissent, comme par exemple la gestion des pannes très fréquentes.

L'avantage principal de cette architecture est la performance, la proximité et les liaisons performantes entre des divers éléments, permettant principalement de réduire le coût des communications entre les différents processeurs. Le principal inconvénient de cette architecture est son coût, aussi bien d'acquisition (le matériel et l'environnement) que de fonctionnement (consommation et maintenance). Cette architecture est principalement utilisée pour l'exécution d'applications nécessitant une grande quantité de ressources sur un laps de temps assez court. De plus elles permettent d'exploiter le parallélisme de certaines applications au niveau matériel.

2.1.2 Clusters

Les clusters, ou grappes de machines, sont une des entités les plus utilisées en calcul numérique parallèle. Le principe de ces grappes de machines est apparu vers les années 1990. C'est à cette époque que les ordinateurs « classiques » devinrent puissants et peu chers. En effet, il suffit de réunir plusieurs

1. Le terme « exascale » fait référence au nombre d'opérations que peut effectuer un supercalculateur en une seconde. On parle ici de 10^{18} opérations. Les enjeux actuels se portent sur le « petascale » qui représente 10^{15} opérations par seconde.

machines en réseau, afin d'augmenter les capacités de stockage, de mémoire et de calcul. La figure 2.2 montre une telle architecture.

FIGURE 2.2 – Une grappe de serveurs, ou cluster

Dans ce type d'architecture, on peut trouver différents types de réseaux, comme du réseau ethernet mais aussi du réseau InfiniBand dont le principe est de fournir, en théorie, une bande passante infinie. On peut aussi trouver des machines servant uniquement de serveurs de stockage ou bien des machines dites « frontales » dont le rôle est de gérer les machines de calcul – en principe, les utilisateurs n'accèdent pas aux machines de calcul mais à une frontale qui se charge ensuite de distribuer les calcul sur les machines constituant la grappe.

Les avantages d'une telle architecture sont le faible coût d'acquisition par rapport à un supercalculateur et ses facilités de maintenance et d'évolution. L'inconvénient majeur est sa limite d'extension, qui est généralement d'une centaine de machines. Cette limite est imposée par les coûts de consommation électrique et les besoins en climatisation pour refroidir les machines. Cette architecture permet d'exécuter des plus longues applications demandant de nombreuses ressources.

2.1.3 Grilles

Les grilles, ou les architectures de clusters distribués, peuvent être considérées comme l'évolution des clusters. En effet, les clusters sont des architectures puissantes, mais assez limitées dans leur évolution. Un cluster ne peut être « agrandi » indéfiniment, principalement pour des raisons de place et de coût de fonctionnement et de maintenance.

Pour pallier ces contraintes, les grilles permettent de relier divers clusters et supercalculateurs, pouvant être géographiquement éloignés, afin d'agréger les ressources de calcul de diverses entités, telles que des entreprises ou des laboratoires de recherche par exemple.

La figure 2.3 représente la grille d'expérimentation Grid'5000 [2], qui est une plateforme de calcul française composée de clusters distribués, dédiée à la recherche en informatique. Comme on peut le remarquer sur la figure, cette grille interconnecte les ressources de plusieurs laboratoires de recherche, 9 en l'occurrence. Chacun de ces centres possède plusieurs clusters internes. Actuellement, cette plateforme regroupe 25 clusters, ainsi répartis sur les 9 sites, pour un total de près de 1500 machines donnant accès à 7468 cœurs de calcul. Chaque machine possède entre 2 et 48Go de mémoire RAM. Tous les sites sont reliés entre eux via le réseau RENATER [4], le « Réseau national de télécommunications pour

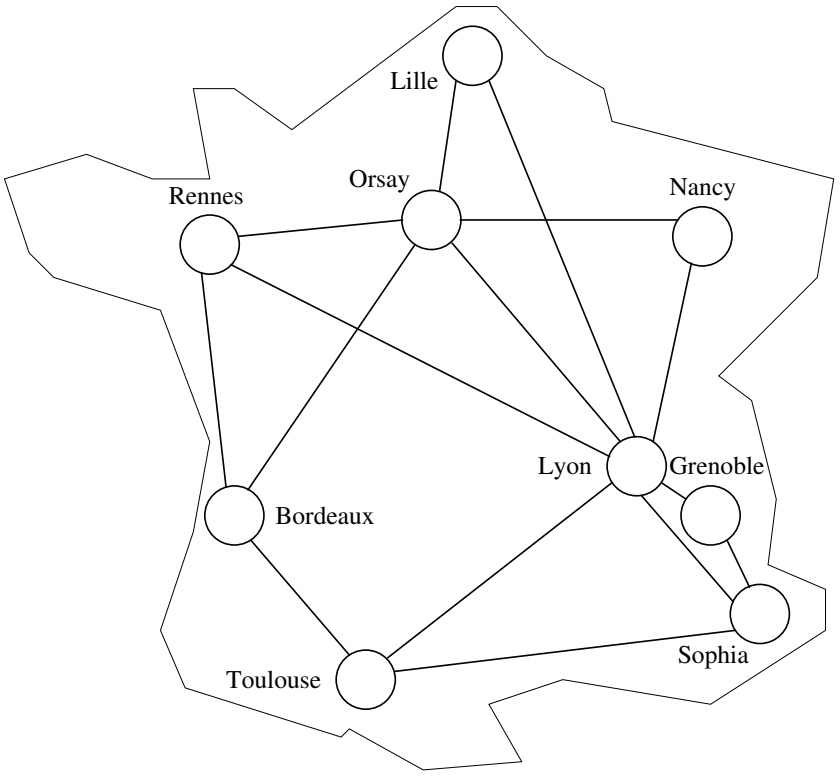

FIGURE 2.3 – La grille d'expérimentation Grid'5000

la technologie, l'enseignement et la recherche ». Ce réseau procure des liens en fibre optique entre les différents centres de recherche, avec une large bande passante et une assez faible latence. Les clusters à l'intérieur d'un même site sont reliés entre eux via des réseaux à très haut débit et faible latence (réseau gigabit, infiniband...).

Ce genre d'architecture permet de fédérer beaucoup de ressources, aussi bien d'un point de vue puissance de calcul que d'un point de vue de la capacité de stockage. Le principal avantage de cette architecture est de proposer une nombre très important de machines. Il devient alors plus aisé d'exécuter un plus grand nombre d'applications et de partager les ressources entre les différents utilisateurs. Pour les applications parallèles comprenant un grand nombre de tâches et/ou utilisant des données très volumineuses, leur exécution est facilitée par la fédération des ressources. Cependant, ceci engendre des complications telle que la perte de performance au niveau réseau. En effet, il suffit que pour l'exécution d'une application deux sites distants doivent être utilisés et que ceux-ci ne soient pas directement reliés entre eux. Ceci augmente la latence en allongeant les chemins de communication. Ce cas de calcul induit une part d'hétérogénéité, aussi bien au niveau des machines de calcul qu'au niveau réseau. Concernant l'hétérogénéité des machines, il est inconcevable de forcer toutes les entités à acheter le même matériel au même moment. Chaque entité a ses besoins propres et met à disposition de la communauté ses ressources. Ceci peut engendrer des pertes de performance, dans le cas où des machines plus « lentes » sont utilisées. Concernant l'hétérogénéité du réseau, tous les liens entre les sites ne sont pas forcément aussi performants, cette performance étant dépendante de la distance entre les sites, du nombre de relais entre eux, et bien sûr de la charge des liens. C'est pourquoi, aux vues de ces problèmes, de nombreux axes de recherche tentent de les résoudre ; le modèle itératif asynchrone en est un bon exemple. Tout comme pour les clusters, cette architecture est dédiée aux applications dont le temps d'exécution est assez long. Le fait de fédérer de plus nombreuses ressources permet une répartition plus aisée de celles-ci aux divers utilisateurs.

2.1.4 Calcul volontaire

La dernière plateforme de calcul parallèle très utilisée est appelée plateforme de calcul volontaire, ou « desktop grid » [25]. L'essence même de cette plateforme est d'utiliser une architecture déjà existante, mais qui n'est pas dédiée au calcul scientifique. Le but ici est de ne pas investir dans des plateformes comme celles décrites dans les précédentes sections (les supercalculateurs, les clusters et les grilles), mais d'utiliser les machines de travail présentes dans les entreprises ou dans les institutions, mais également les ordinateurs personnels. L'investissement initial est donc drastiquement réduit ainsi que les coûts de maintenance. De même, l'extension d'une telle architecture est très facile.

À titre d'exemple, on peut citer le projet Seti@Home [10], comme l'un de précurseur les plus connus, dont le principe est détaillé en section 1.2.1. La figure 2.4 présente un schéma d'une telle architecture.

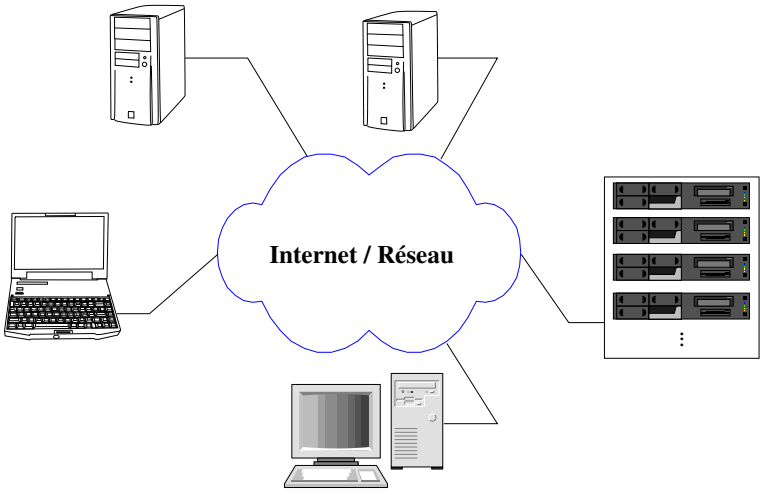

FIGURE 2.4 – Plateforme de calcul volontaire, ou « desktop grid »

Comme on peut le voir sur la figure 2.4, n'importe quelle ressource (ordinateur de bureau, ordinateur portable, serveur, téléphone portable...) connectée au réseau Internet peut faire partie d'une telle architecture. Il existe une variante de cette architecture, ayant pour vocation d'exécuter des applications avec des dépendances, impliquant donc des communications, qui est appelée « enterprise desktop grid » – expression pouvant être traduite par « grille d'ordinateurs de bureau ». Le but de ce type de plateforme est de constituer un cluster en utilisant les ressources disponibles au sein d'une entité, ou institution. Par exemple, pour les besoins de nos travaux sur la création d'une plateforme de calcul utilisant des machines virtuelles, décrits dans la partie III de ce document, les machines à disposition des étudiants du département Informatique de l'IUT de Belfort-Montbéliard forment une telle architecture.

Le principal avantage de ces architectures est le faible investissement nécessaire, car elles utilisent le matériel existant. Cependant bon nombre d'inconvénients sont à prendre en compte. Un des principaux inconvénients est la disponibilité des machines constituant la plateforme. En effet, comme ce sont des machines mises à disposition par les utilisateurs, lorsque ceux-ci travaillent, il ne faut pas qu'ils soient importunés par les calculs. Le plus souvent, pour remédier à ce problème, le calcul est soit arrêté soit mis en pause, puis déplacé et redémarré sur une autre machine, qui est elle, disponible. Un autre problème, qui est commun aux grilles, est l'hétérogénéité des machines, qui peuvent être de différentes générations. Enfin un dernier problème concerne le réseau, qui lui aussi peut être différent, par exemple à très haut débit dans une salle et à plus faible débit entre deux bâtiments, sans parler des problèmes de gestion du réseau concernant le routage et les infrastructures mises en place par les administrateurs de l'entité.

2.2 Environnements d'exécution

Après avoir présenté les différents types d'algorithmes parallèles et les plateformes d'exécution, cette section a pour objectif de présenter les principaux environnements de programmation et d'exécution utilisés pour le calcul parallèle. Ces environnements permettent le développement d'applications parallèles, mais aussi permettent de les exécuter sur les architectures décrites précédemment. Certains environnements ne sont conçus que pour exécuter des applications, laissant ainsi au programmeur la liberté du choix du langage d'implémentation, en imposant néanmoins quelques contraintes de fonctionnement de l'environnement.

Toutes les plateformes permettant le déploiement d'applications parallèles sur des architectures distribuées existantes ne sont pas présentées ici, comme par exemple Condor [64], Globus [32], gLite [46], et bien d'autres. Celles présentées ici sont des exemples de plateformes dédiées aux diverses architectures présentées dans la première partie du chapitre. L'accent est principalement mis sur les environnements qui ont été utilisés au cours de cette thèse et qui ont fait l'objet de contributions.

Dans un premier temps, nous présentons l'un des environnements les plus connus et utilisés, à savoir la bibliothèque MPI. Ensuite sont présentés plus en détails les environnements ProActive, XtremWeb et XtremWeb-CH ; le premier étant un environnement complet, permettant l'implémentation et l'exécution d'applications, les deux autres étant uniquement des environnements d'exécution. Enfin, en dernière section, sont présentés les environnements Jace, JaceP2P et JaceP2P-V2 – ce dernier ayant été utilisé pour les travaux présentés dans la partie II.

2.2.1 MPI

MPI [50], signifiant « Message Passing Interface », littéralement interface de passage de messages, est une norme définissant une bibliothèque de fonctions, permettant de paralléliser des programmes suivant le paradigme de passage de messages. Le principe est de pouvoir exploiter des machines multiprocesseurs, telles que les supercalculateurs, ou des machines distantes, comme les clusters ou les grilles de calcul. Cette norme est apparue en 1993 et une deuxième version a vu le jour en 1997. De par sa popularité et sa forte utilisation, cette norme est devenue un standard de communication pour les applications parallèles. Des bibliothèques mettant en œuvre cette norme existent pour quasiment toutes les architectures matérielles et sont disponibles pour de nombreux langages de programmation. Ces bibliothèques sont utilisées pour exécuter des applications parallèles sur tous types d'architectures, mais elles sont principalement utilisées sur des architectures distribuées telles que les clusters et les grilles de calcul.

Pour exécuter une application à l'aide d'une bibliothèque MPI, il faut appeler le lanceur de la bibliothèque. Cette entité sera en charge de créer des démons de calcul et d'en assurer la gestion, jusqu'à la fin de l'exécution de l'application. Ce lanceur prend en paramètre l'exécutable de l'application, une liste de machines sur lesquelles seront effectués les calculs, ainsi que les paramètres éventuels de l'application. À son lancement, MPI va donc créer des « démons » qui sont en charge de la gestion des communications entre les différentes tâches ; tous ces démons sont en liaison avec le lanceur. MPI utilise alors un communicateur global, qui permet aux processus de communiquer entre eux. Il est à noter ici que si deux processus veulent communiquer, ils doivent obligatoirement faire partie du même communicateur – ce mécanisme permet d'isoler logiquement les communications si plusieurs applications s'exécutent en même temps sur une même machine. Ensuite, le programmeur doit utiliser les fonctions de la bibliothèque pour que les processus s'échangent des données durant l'exécution. La figure 2.5 décrit quelques unes de ces fonctions. La première fonction représente une opération de « broadcast » qui permet d'envoyer des données à toutes les tâches en une seule opération ; la seconde représente les mécanismes d'envoi/réception de messages ; la troisième représente l'opération

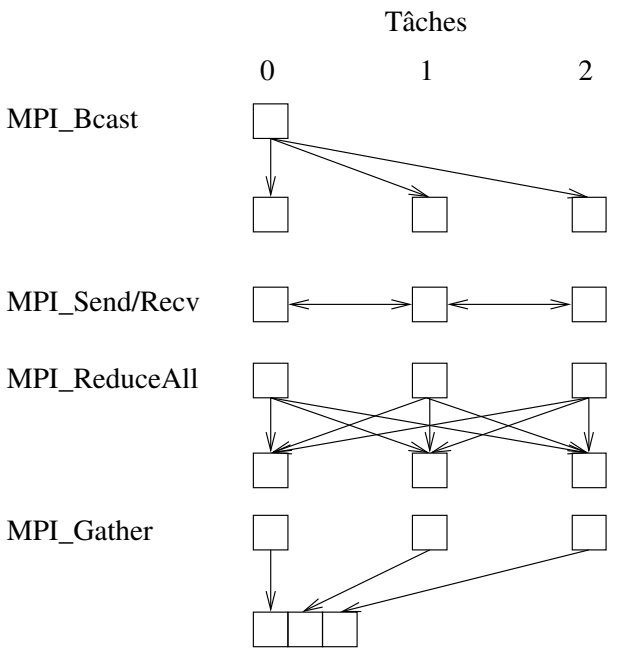

FIGURE 2.5 – Quelques fonctions de base de la norme MPI

de réduction globale permettant de rassembler des résultats sur toutes les tâches en même temps ; et enfin la dernière représente l'opération de collecte de données par une tâche.

Il existe bien d'autres fonctions, mais il est inutile d'être exhaustif ici. D'autres fonctions, celles-ci de gestion, sont à utiliser pour initialiser le communicateur, l'environnement, mais également pour terminer proprement une application.

Les implémentations classiques de MPI sont faites en C/C++ et Fortran, notamment avec les bibliothèques MPICH [37] et OpenMPI [33] par exemple. Mais d'autres bibliothèques existent pour les langages Python, OCaml ou encore Java. Ces bibliothèques sont des adaptations de la norme. Il existe également des versions intégrant des mécanismes de tolérance aux pannes, telles que MPICH-V [20] qui applique la méthode de sauvegarde des messages, ou P2P-MPI [36] qui utilise la réplication des tâches.

2.2.2 ProActive

ProActive [11] est un environnement de programmation et d'exécution, qui est dédié à la réalisation de programmes reposant sur des algorithmes totalement synchrones. Il est écrit en Java et les applications sont elles aussi écrites dans ce langage, avec cependant la possibilité d'exécuter une application écrite en MPI, via l'emploi de primitives dédiées. Cet environnement est essentiellement destiné à l'exécution d'applications parallèles sur des grilles de calcul.

ProActive [2] a été développé à l'INRIA de Sophia Antipolis, et depuis, une société distribue le projet pour permettre de sortir du cadre de la recherche et toucher plus de domaines (comme l'industrie) ; cette société s'appelle ObjectWeb [3].

Le principe de fonctionnement de l'environnement est calqué sur le modèle MPI, avec le déploiement de démons de gestion sur les machines de calcul. La configuration du déploiement de l'architecture est réalisée à l'aide d'un « descripteur de déploiement » qui a la forme d'un fichier XML.

2. Le site Internet du projet : `http://proactive.inria.fr`
3. Le site Internet : `http://www.objectweb.org`

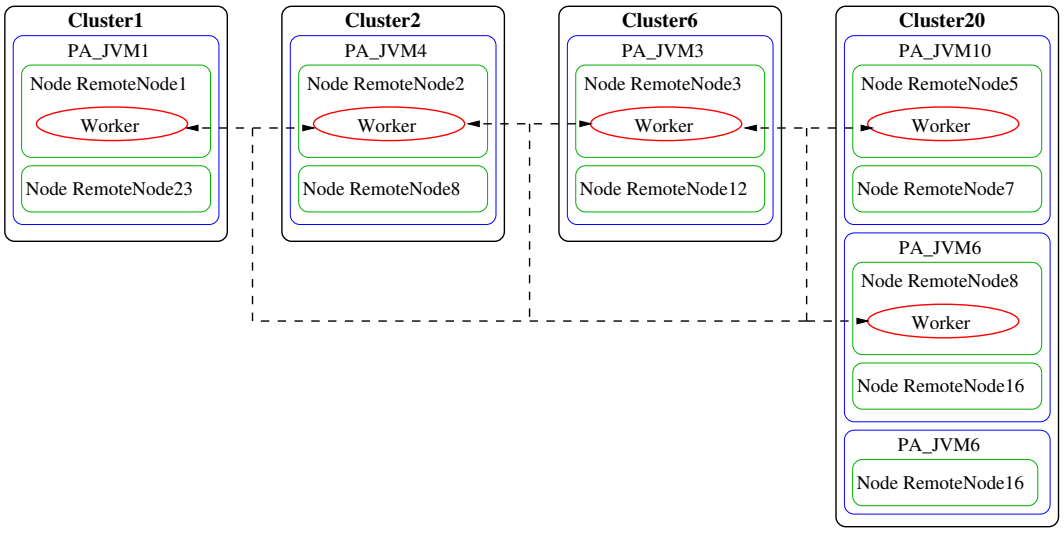

FIGURE 2.6 – Architecture de ProActive

En ce qui concerne l'architecture même de ProActive, elle est basée sur l'exécution d'une ou plusieurs machines virtuelles Java par noeud de calcul. Comme on peut le voir sur la figure 2.6 , il peut y avoir sur une même machine physique plusieurs machines virtuelles. Toutes les tâches communiquent entre elles via le protocole RMI, et pour connaître l'adresse de leur destinataire, elles utilisent des primitives renvoyant la liste des « workers » sous forme d'un groupe de travail.

Le modèle de communication de ProActive ne permet pas, de par son principe, de concevoir des applications reposant sur des algorithmes itératifs asynchrones. Cependant, les mécanismes en place apportent une certaine souplesse au niveau des communications. En effet, lorsque deux machines échangent des données, elles ne sont plus contraintes de se synchroniser avant d'effectuer leur échange. Ceci provient du fait que les appels via le protocole RMI ont été modifiés. En Java classique, ces appels se font de manière synchrone, c'est-à-dire que l'objet appelant se met en attente de disponibilité de l'objet appelé – il attend en fait que celui-ci soit à son écoute pour pouvoir accéder à sa requête. Quand les deux objets sont prêts à échanger, l'objet appelé exécute la demande de l'appelant – généralement une exécution d'une fonction – et lui renvoie un résultat, ou exécute une action. Avec ProActive, ces concepts sont modifiés, puisque les appels RMI ont été rendu « asynchrones ». C'est-à-dire que l'objet appelant n'attend plus que l'objet appelé soit à son écoute pour demander une exécution de fonction, ainsi que son éventuel retour. Cependant, il attendra l'arrivée de la réponse dans le cas où il doit utiliser cette valeur de retour (on parle ici d'« objet futur » qui est en fait une représentation de l'objet qui doit être renvoyé), et donc il attendra que l'appelé ait terminé d'exécuter la fonction. Ainsi, ProActive seul ne permet pas de réaliser des programmes reposant sur des algorithmes asynchrones, car aucun mécanisme n'est prévu à cet effet. Cependant, nous avons mené des travaux visant à la création d'une bibliothèque additionnelle, AIL-PA [24] (pour « Asynchronous Iterative Library for ProActive »), permettant le développement et l'exécution d'algorithmes itératifs asynchrones avec ProActive. Elle ajoute des mécanismes de communications supplémentaires, avec des résultats très intéressants.

Un des inconvénients majeurs de ProActive est la taille mémoire qu'il nécessite lors de l'exécution d'un programme. En effet, beaucoup de classes sont chargées en mémoire – c'est un environnement volumineux, offrant beaucoup de fonctionnalités, comme par exemple la « migration d'objets[4] », le « monitoring » de l'exécution via une interface montrant les différentes interactions entre les tâches, ou la possibilité d'exécuter du code écrit en MPI via des primitives de l'environnement.

4. La « migration d'objets » est le fait de pouvoir transporter l'exécution d'un programme d'une machine à une autre.

2.2.3 XtremWeb / XtremWeb-CH

Les deux plateformes XtremWeb [30] et XtremWeb-CH [7] sont des plateformes dédiées au calcul volontaire, ou « volunteer computing » en anglais, dont les principes sont décrits en section 2.1.4. Ces plateformes sont plutôt dédiées au modèle de calcul parallèle sans communication.

XtremWeb

Le projet XtremWeb, développé au LRI à Orsay, a pour but de fournir une plateforme de calcul volontaire. Cette plateforme « open source » (à sources ouvertes et libres) peut être déployée en entreprise, dans des centres de recherche, en utilisant aussi bien des clusters, des grilles, tout en y ajoutant des ressources externes, comme des ordinateurs personnels reliés à Internet. Elle utilise les temps d'inactivité des machines afin d'effectuer des calculs scientifiques.

Son architecture repose sur trois éléments : un client, un coordinateur et des démons. Un quatrième élément peut être ajouté, qui est un entrepôt pour stocker les données et les exécutables. C'est une architecture classique de calcul volontaire. La figure 2.7 décrit l'architecture de XtremWeb. Chaque élément a un rôle précis :

- le coordinateur : c'est l'élément central de l'architecture. C'est cette entité qui centralise aussi bien les applications que les données associées. Comme les clients et les démons ne peuvent pas communiquer directement, tout passe par lui. Il est aussi en charge de la surveillance des démons. En effet, lorsque l'un d'eux tombe en panne, il redistribue sa tâche à une autre machine disponible. Comme on peut le remarquer sur la figure 2.7, tout passe par le coordinateur ;
- le client : il est exécuté par un utilisateur qui veut effectuer un calcul. Il sert d'interface avec la plateforme. L'utilisateur peut ainsi envoyer le ou les exécutables de son application ainsi que les données associées au coordinateur. Ensuite, il indique combien de tâches composent l'application ;
- le démon : lors de son lancement, il se connecte au coordinateur. Périodiquement, il lui envoie une requête afin d'obtenir une tâche à effectuer. S'il est sélectionné, il récupère alors un exécutable et les données correspondantes au calcul. Une fois le calcul terminé, il renvoie les résultats au coordinateur, sur lequel le client peut les récupérer.

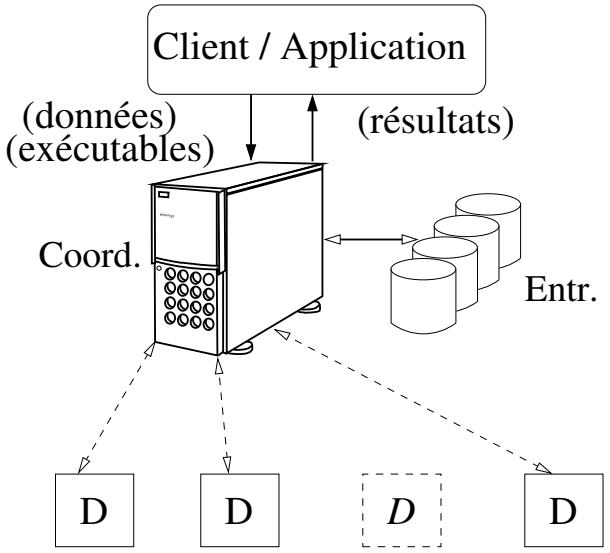

FIGURE 2.7 – Architecture de XtremWeb

Afin d'assurer le passage à l'échelle et une forme de tolérance aux pannes, le coordinateur peut être répliqué. Ainsi la charge peut être répartie, notamment celle de surveillance des machines de calcul.

De par ces mécanismes de communication des données et des applications, XtremWeb peut traverser des pare-feu, lui permettant ainsi de fédérer beaucoup de machines. Cependant, l'inconvénient de cette méthode est que les machines ne peuvent pas échanger de messages entre elles. Il n'est donc pas envisageable d'exécuter des applications possédant des dépendances entre les tâches, nécessitant alors des échanges de données en cours de calcul.

XtremWeb-CH

En conséquence du principal inconvénient de XtremWeb, qui est que les tâches de calcul ne peuvent pas communiquer entre elles, est né XtremWeb-CH. Cette plateforme peut être vue comme une amélioration de la précédente. Celle-ci a été réécrite en Java par une équipe de l'HES-SO, la Haute École Spécialisée de Suisse Occidentale, à Genève. Tout en conservant les mécanismes de base de XtremWeb, une couche communication a été ajoutée. La figure 2.8 présente l'architecture de XtremWeb-CH.

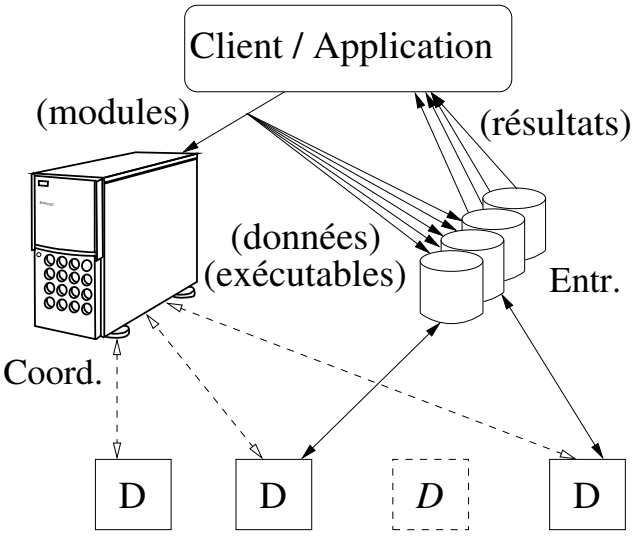

FIGURE 2.8 – Architecture de XtremWeb-CH

Cette fois, l'entrepôt devient un élément essentiel. Cette entité sert de système de stockage pour la plateforme. C'est sur cette machine que sont donc déposés les exécutables et les données, mais aussi les résultats. Cet entrepôt sert aussi aux communications entre les différentes tâches de calcul. Le principe est qu'une tâche communique avec les autres par le biais de fichiers de données. Comme on peut le voir sur la figure 2.8, cette fois les clients et les démons peuvent échanger directement avec l'entrepôt. Ceci permet de décharger le coordinateur des tâches de transfert. Il est à noter que XtremWeb-CH permet d'utiliser plusieurs entrepôts, afin de créer des redondances de données (pour la tolérance aux pannes), mais aussi de placer des entrepôts plus près des démons, afin d'optimiser les transferts.

Avec XtremWeb-CH, la nature des applications change elle aussi. En effet, une application n'est plus forcément représentée par un seul exécutable, mais par des modules. Un module peut être défini comme une unité de calcul au sein d'une application. Prenons l'exemple d'une application de calcul indépendant, comme celle représentée par la figure 1.1. La première partie, celle du découpage, peut être un module, le calcul un autre module, et enfin l'assemblage le dernier module. Dès lors, on remarque que certains modules ont besoin des données résultats d'autres modules pour pouvoir s'exécuter. Ces données transitent donc soit directement entre les machines qui doivent exécuter les tâches si elles peuvent communiquer directement, soit en utilisant les entrepôts. Un module est un ensemble composé de données et d'exécutables – un module peut contenir un même programme, mais compilé pour différentes plateformes (le choix est fait par le démon lors de l'exécution de la tâche).

Une application est donc un ensemble de modules pouvant s'exécuter sur différentes plateformes, et cette fois ayant des communications entre eux. Cependant ce modèle oblige le programmeur à tenir compte de ces mécanismes pour effectuer des communications. La « vraie » application est en fait un programme de gestion des appels aux modules sur la plateforme, l'application originale étant découpée en modules. Cette approche permet d'utiliser des machines qui ne sont pas accessibles directement depuis un réseau externe, notamment à cause de restrictions de sécurité comme par exemple l'utilisation d'un pare-feu.

Une telle approche des communications, par échanges de fichiers, peut entraîner des pertes de performances pour des applications ayant des échanges fréquents. Nous avons pu porter une application, Neurad [16], sur une plateforme XtremWeb-CH, et les résultats obtenus sont plutôt encourageants [8].

2.2.4 Jace / JaceP2P & JaceP2P-V2

Dans cette section sont présentés trois environnements de programmation et d'exécution d'applications, dédiés au modèle des algorithmes itératifs asynchrones (IACA). Ces trois environnements ont été développés au sein de l'équipe AND (Algorithmique Numérique Distribuée) du laboratoire LIFC (Laboratoire d'Informatique de Franche-Comté), où cette thèse a été réalisée. Dans un premier temps nous présentons Jace, qui permet à la fois de programmer et d'exécuter des algorithmes itératifs synchrones et asynchrones, sur des plateformes distribuées. Dans un second temps est décrit JaceP2P, qui est une évolution de Jace, intégrant principalement des mécanismes de tolérance aux pannes. Enfin, nous présentons JaceP2P-V2, qui est une amélioration de JaceP2P, notamment concernant la tolérance aux pannes, mais surtout le passage à l'échelle. Il est à noter ici que les environnements JaceP2P et JaceP2P-V2 ne permettent la mise en œuvre et l'exécution que des algorithmes itératifs asynchrones.

Jace

Jace [15], signifiant « Java Asynchronous Computation Environment » (environnement de calcul asynchrone en Java), est le premier environnement dédié à la programmation et à l'exécution d'algorithmes itératifs asynchrones sur des architectures distribuées. Bien que dédié au modèle itératif asynchrone, Jace permet aussi l'exécution d'algorithmes itératifs synchrones.

Jace se décompose en deux parties :

- une bibliothèque de méthodes : cette partie sert notamment au développement d'applications itératives. Le modèle de cette bibliothèque reprend les principes de la norme MPI. Pour des raisons de portabilité du code elle est écrite en Java ;
- une plateforme d'exécution : cette partie, décrite plus en détails dans cette section, est déployée sur l'architecture d'exécution. Elle s'occupe de gérer le bon déroulement des exécutions.

La figure 2.9 présente l'architecture de la plateforme d'exécution. On remarque qu'elle est basée sur les trois composants suivants :

- les démons : les démons sont exécutés sur chaque machine de l'architecture. Ils contrôlent l'ensemble de l'environnement en initialisant les « workers » qui sont les tâches exécutant le programme, en surveillant les machines, en collectant les résultats... Pour permettre une bonne évolution de l'architecture et un déploiement à grande échelle (sur grille notamment), ils sont organisés sous forme d'arbre binomial ;
- le lanceur : cette entité permet d'exécuter une application, ou une tâche [5]. Il prend en charge une liste de paramètres : l'application, les paramètres de l'application, le nombre de tâches à

5. Une tâche est une unité de calcul qui est exécutée comme un « thread ».

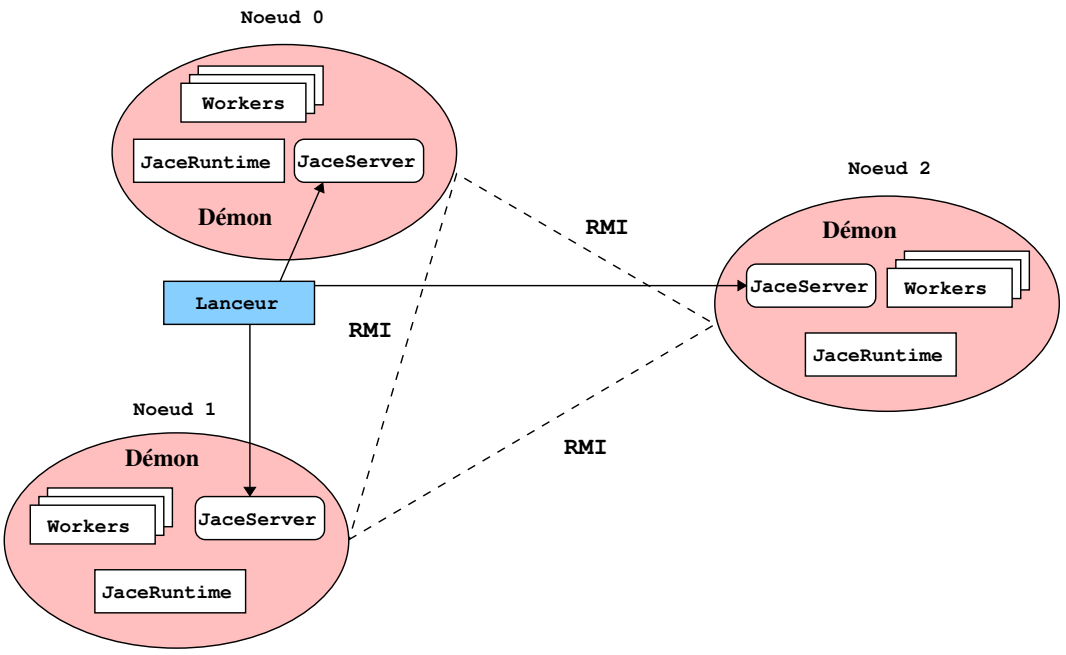

FIGURE 2.9 – Architecture de Jace

exécuter, la liste des démons participants, et l'algorithme de distribution des données (« Round Robin » tour à tour, « best effort » non ordonné). Quand un démon reçoit un message du lanceur, il le transmet à tous ses voisins (dans l'arbre binomial) et lance un « worker » pour charger et exécuter les tâches demandées ;

- les « workers » : comme énoncé précédemment, ils sont créés par les démons, suivant une requête d'un lanceur. Cette entité est structurée en deux couches :
  - la couche application : cette couche permet l'exécution des tâches et la détection de la convergence de l'algorithme. En ce sens, le processus de détection de convergence est ainsi caché à l'utilisateur. Il est à noter que pour utiliser au mieux les machines multi-cœurs et multi-processeurs, cette couche permet au démon d'exécuter plusieurs tâches, ou de paralléliser la tâche (si possible) ;
  - la couche communication : cette couche est le point central de la plateforme Jace. Elle est composée de deux « threads » (**sender** pour les envois de messages et **receiver** pour la réception) et d'un ensemble de files de messages. La gestion de ces différentes files diffère suivant le mode d'exécution – synchrone (pour les algorithmes ISCS et ISCA) ou asynchrone (pour les algorithmes IACA). Jace offre la possibilité de choisir le protocole de communication parmi : les *« sockets » TCP/IP*, le protocole *NIO* (« New Input/Output » – nouvelles entrées/sorties) et le protocole *RMI*.

Chaque tâche va échanger des messages avec ses dépendances et va utiliser pour cela la méthode de transmission de données par passage de messages. Jace propose un mécanisme permettant de gérer l'envoi et la réception de ces messages, via une file d'attente. Cette file, suivant le modèle d'algorithme utilisé, synchrone ou asynchrone, va gérer différemment les messages : en mode synchrone, tous les messages reçus sont stockés puis restitués un à un au destinataire ; en mode asynchrone, on ne trouve dans la file de messages, aussi bien en envoi qu'en réception, que le dernier message correspondant à un triplet composé des champs destinataire/émetteur et d'un « tag ». Le tag permet de définir dans quelle séquence de l'algorithme se situe l'échange de données. Le principe ici est de ne prendre en compte que la dernière version des messages, ceci afin de ne garder que les données les plus à jour, ce qui permet de progresser plus rapidement vers un état de convergence.

Concernant la convergence, le mécanisme mis en place dans Jace est une détection centralisée. Lorsqu'un démon détecte la convergence locale, il envoie un message de convergence à l'entité en charge de la détection de la convergence globale. Si tous les démons ont envoyé un message de convergence, la convergence globale est alors détectée. Cependant, comme indiqué en section 1.3.2, quelques précautions sont à prendre. Pour rappel, puisqu'une tâche ne reçoit pas à chaque itération des mises à jour, ceci peut mener à la détection d'une fausse convergence locale. Dans Jace, une protection est mise en place. En effet, lorsqu'un démon détecte une convergence locale, il laisse la tâche effectuer un nombre prédéfini d'itérations, afin de s'assurer de la véritable convergence. Ce mécanisme a l'avantage de réduire les risques de détection d'une fausse convergence, mais a aussi l'inconvénient d'augmenter significativement le temps de détection d'une vraie convergence, augmentant par la même le temps d'exécution de l'application.

Bien que performante, cette plateforme présente quelques inconvénients, qui prennent de l'importance lors d'une utilisation sur des environnements tels que les grilles de calcul. Les principaux inconvénients sont la non tolérance aux pannes, car il n'y a pas de mécanisme de détection de panne, et le non passage à l'échelle. En effet, tout est centralisé par le lanceur, qui doit supporter toute la charge de surveillance et d'exécution, et ce peu importe le nombre de machines utilisées. Néanmoins, la plateforme Jace est simple à déployer et présente de bonnes performances.

JaceP2P

L'environnement JaceP2P [14], acronyme de « Java Asynchronous Computation Environment for Peer-to-Peer architectures », a été créé pour pallier les défauts de Jace. Cet environnement a également été développé par l'équipe AND. La motivation de JaceP2P est de fournir une plateforme pair-à-pair, permettant d'utiliser les ressources disponibles qui ne sont pas obligatoirement dédiées au calcul – reprenant ainsi l'esprit des plateformes de calcul volontaire. Comme la plateforme vise des architectures volatiles, la principale évolution par rapport à Jace est la mise en place d'une politique de tolérance aux pannes. Une autre différence avec l'environnement Jace provient du fait que JaceP2P ne permet d'exécuter que des applications itératives asynchrones. Bien que pensé pour les environnements pair-à-pair, JaceP2P peut aussi très bien s'exécuter sur des architectures distribuées telles que les clusters ou les grilles de calcul.

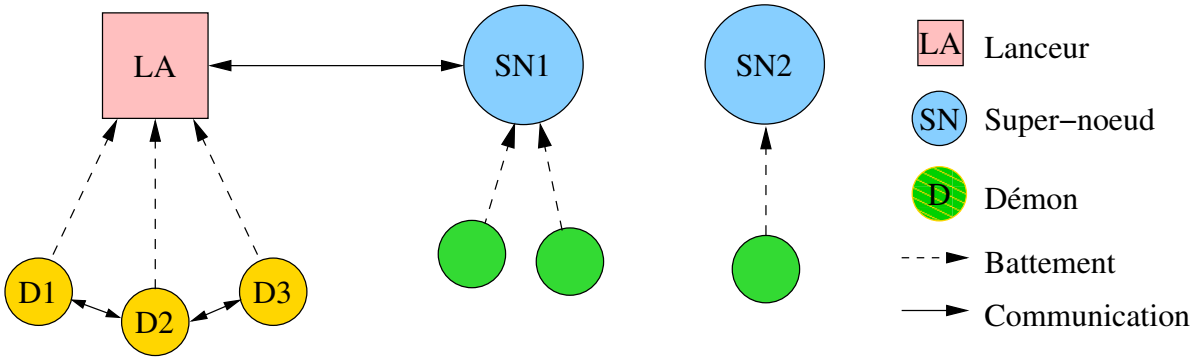

FIGURE 2.10 – Architecture de JaceP2P

La figure 2.10 présente l'architecture de la plateforme d'exécution JaceP2P. Comme on peut le voir, l'architecture est composée de trois entités :

- le super-nœud : cette entité, représentée par un cercle sur la figure, est le point d'entrée de la plateforme. Les machines voulant contribuer à la plateforme se connectent sur un super-nœud. Chaque super-nœud maintient un registre des démons qui lui sont connectés – ce registre

contient l'adresse IP des démons disponibles (ceux qui ne participent pas à l'exécution d'une application). Le super-nœud surveille les démons connectés, et lorsqu'il ne reçoit plus de « message de battement de cœur », appelé « heartbeat », d'un démon, celui-ci est retiré du registre ;
- le lanceur : cette entité, représentée par un rectangle, reprend les principales fonctions de celui de Jace. Quand un utilisateur veut exécuter une application il crée un lanceur avec pour paramètres le nombre de machines nécessaires à l'exécution de l'application, la localisation des fichiers de l'application, et éventuellement des paramètres pour l'application. Une fois créé, le lanceur contacte un super-nœud pour réserver le nombre de machines requises. S'il n'y a pas assez de machines disponibles sur le super-nœud contacté, ce dernier demande à un autre super-nœud d'autres machines. Les démons réservés sont retirés des registres des super-nœuds et une liste est transmise au lanceur. Quand celui-ci reçoit cette liste, il crée une tâche pour chaque démon, et commande l'exécution de celles-ci. Le lanceur leur communique aussi le registre afin que les démons puissent communiquer entre eux. Le lanceur a aussi la charge de surveiller les démons participant au calcul ; lors de leur réservation, les démons envoient leurs messages de heartbeat au lanceur et non plus à leur super-nœud d'attache. Lorsque le lanceur détecte la panne d'une machine, il contacte un super-nœud pour réserver une machine remplaçante. Lors de la réception de celle-ci, le lanceur l'initialise et le nouveau démon récupère la dernière sauvegarde et rejoint ainsi le calcul en cours. La dernière fonction du lanceur est de gérer la détection de la convergence globale. Ici, la détection de la convergence globale est également centralisée, comme dans Jace. Ceci vient du fait que le lanceur est considéré comme s'exécutant sur une machine sûre ;
- le démon : cette entité est celle en charge du calcul. Lors de son lancement, le démon se connecte à un super-nœud, et attend d'être utilisé pour effectuer un calcul. Durant l'exécution d'une application, tous les démons peuvent communiquer directement entre eux, et effectuent régulièrement leurs sauvegardes sur leurs voisins de calcul. Une fois l'application terminée, chaque démon se reconnecte sur un super-nœud.

La différence la plus importante face à Jace est que JaceP2P intègre des mécanismes de tolérance aux pannes. En effet, le lanceur est en charge de surveiller les démons. Par un mécanisme de heartbeat, le lanceur peut détecter la panne d'un démon, et ainsi appliquer les fonctions de remplacement. Pour que ceci soit possible, il a été mis en place un système de sauvegarde des tâches, basé sur le modèle des sauvegardes non coordonnées, décrites en section 1.4.4. Pour rappel, l'utilisation de cette méthode est possible de par le fait qu'il n'y a pas de synchronisation entre les tâches. L'utilisateur peut définir la fréquence des sauvegardes. Les nœuds de sauvegarde sont sélectionnés parmi les voisins de calcul. À chaque sauvegarde, celle-ci est envoyée à un voisin différent. Le mécanisme utilisé pour la sélection de ce voisin est celui du « tourniquet » (*round robin*). Ceci permet de répartir les sauvegardes au fil du temps, afin d'éviter de perdre toutes les sauvegardes situées sur une seule machine – cela ajoute un degré supplémentaire pour la tolérance aux pannes.

Le principal apport de cette nouvelle version de l'environnement est l'ajout des mécanismes pour la tolérance aux pannes. Cependant, il existe encore quelques limitations à l'exécution de cet environnement dans des environnements distribués, volatils, et comportant un nombre important de machines :

- JaceP2P n'est totalement tolérant aux pannes, car le lanceur est considéré comme étant exécuté sur une machine sûre ;
- la détection des pannes est ici aussi centralisée, sur le lanceur ;
- de même que pour Jace, JaceP2P a une détection de convergence centralisée ;
- comme le montrent les points précédents, JaceP2P possède de trop nombreux mécanismes centralisés, comme le lancement des applications, la détection des pannes, et la détection de la convergence. Ceci limite fortement le déploiement de la plateforme à plus large échelle, et crée des points sensibles pour la tolérance aux pannes de la plateforme complète.

JaceP2P-V2

La dernière version de l'environnement, JaceP2P-V2 [23], apporte des améliorations ainsi que des solutions aux limitations de JaceP2P. Une grande partie du travail a été de décentraliser au maximum tous les mécanismes de gestion de la plateforme.

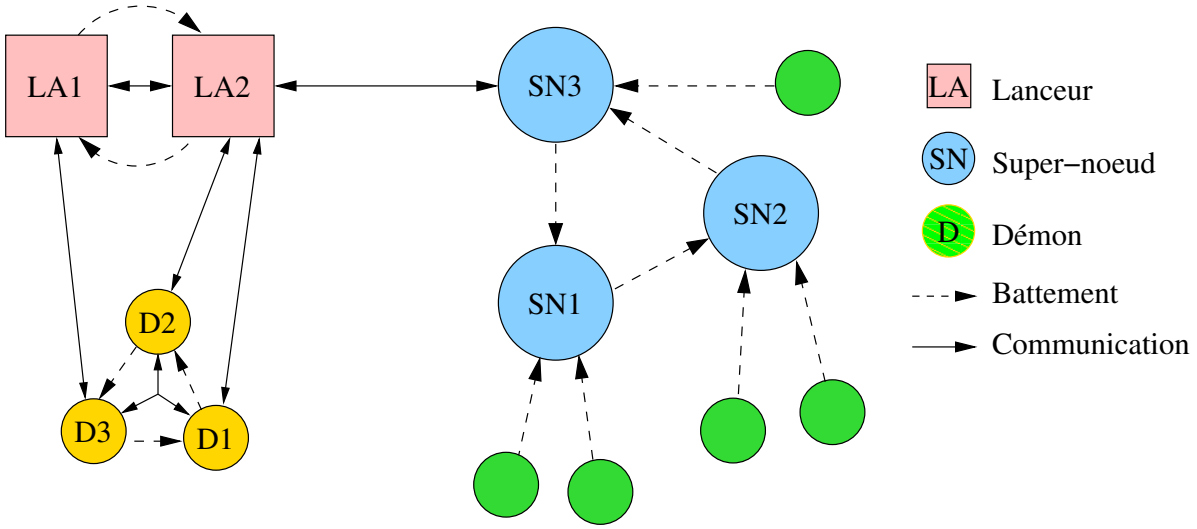

FIGURE 2.11 – Architecture de JaceP2P-V2

La figure 2.11 présente la nouvelle architecture de la plateforme, montrant les mécanismes de tolérance aux pannes. Les changements apportés aux trois entités de base sont :

- le super-nœud : les super-nœuds forment maintenant un réseau circulaire. Cette architecture permet de détecter la panne d'un super-nœud, mais permet également la mise en place d'un mécanisme d'équilibrage de charge. En effet, à tour de rôle, grâce à un jeton, chaque super-nœud peut transférer la surveillance de démons qui lui sont connectés à un autre super-nœud, qui est moins chargé. Dès qu'il a le jeton, il calcule la moyenne du nombre de démons connectés sur chaque super-nœud. Si le nombre de démons qui lui sont connectés est supérieur à cette moyenne, il distribue alors l'excédant aux autre super-nœuds. Pour chaque opération de transfert, la liste des démons est mise à jour sur tous les super-nœuds concernés ;
- le lanceur : cette entité a fait l'objet de nombreuses modifications. Tout d'abord, il a perdu sa fonctionnalité de surveillance des démons, mais également celle de détecter la convergence globale. Ces tâches ont été distribuées sur les démons. Lorsqu'un utilisateur veut lancer une application, il crée un lanceur, qui est alors chargé de réserver des démons auprès des super-nœuds. Au nombre de démons requis pour l'application, le lanceur ajoute quelques démons supplémentaires. Ceux-ci, pour des raisons de tolérance aux pannes, seront des duplicata du lanceur. Ceci permet de tolérer la panne d'un lanceur, mais aussi d'équilibrer la responsabilité des démons de calcul. Ainsi, chaque lanceur est responsable d'un groupe de démons. Les lanceurs forment un réseau circulaire, permettant de détecter la panne d'un lanceur. Lors d'une panne, le lanceur l'ayant détectée contacte un super-nœud afin d'obtenir un démon disponible, qui sera alors transformé en lanceur. Le nouveau lanceur récupérera alors la charge du groupe de démons dont était responsable son prédécesseur, et les démons seront informés de l'adresse de leur nouveau lanceur responsable ;
- le démon : il est toujours en charge de l'exécution d'une tâche. D'autres fonctionnalités lui ont été ajoutées. La première d'entre elles est la détection des pannes des autres démons. Lors du lancement de l'application, les démons forment un réseau circulaire de surveillance. Chaque démon envoie ses messages de heartbeat auprès du démon suivant dans la liste. Si après un certain laps de temps ce dernier n'a pas reçu de message de hearbeat, il informe aussitôt son superviseur

(un lanceur) de la panne. Le lanceur contacte alors un super-nœud pour réserver un démon de remplacement. Une fois ce dernier initialisé, les listes des démons sont mises à jour avec l'adresse du remplaçant. Une autre fonctionnalité déplacée sur le démon est la détection de la convergence globale. Ce mécanisme est maintenant décentralisé [23]. Concernant les sauvegardes, JaceP2P-V2 reprend les mêmes principes que JaceP2P.

Aux vues de ces modifications et améliorations, la plateforme d'exécution JaceP2P-V2 peut être déclarée comme étant totalement tolérante aux pannes. Pour cela, comme il a été montré à l'aide de la figure 2.11, les super-nœuds sont multiples afin de mieux supporter la charge, apportant aussi une redondance permettant de supporter la perte d'un super-nœud. De même, le lanceur, qui est répliqué, a été délesté des tâches de supervision et de détection de la convergence des applications, lui permettant ainsi de ne plus être surchargé. Le fait de dupliquer le lanceur lui permet également de ne plus être un point faible pour la tolérance aux pannes. Ainsi, ce sont les démons qui assurent, de manière décentralisée, la détection des pannes, chacun surveillant un de ses voisins, mais aussi la détection de la convergence globale.

La nouvelle version de la plateforme est donc totalement décentralisée et tolérante aux pannes. Elle est ainsi opérationnelle pour l'exécution d'applications itératives asynchrones dans des environnements distribués et volatils. Cependant, un inconvénient demeure : l'absence d'une politique de placement des tâches de calcul. Ceci est l'objet de la partie II du présent document.

2.3 Conclusion

Plateformes d'exécution

Dans ce chapitre ont été présentées dans un premier temps les architectures les plus couramment utilisées pour le calcul parallèle. Chacune de ces architectures présente des avantages et des inconvénients, et chacune d'entre elles est plus ou moins efficace suivant le type d'application qui est exécutée.

En effet, les supercalculateurs sont très puissants et très efficaces. Cependant, ils restent très chers et peu extensibles. Ils sont dédiés aux applications nécessitant de nombreuses ressources sur un laps de temps assez court. Concernant les clusters, cette architecture est assez simple à mettre en œuvre et peut être une bonne alternative aux supercalculateurs. Cette architecture possède elle aussi des limites au niveau de son extension, aussi bien d'un point de vue de la place occupée, que du coût, que ce soit d'achat, de renouvellement ou bien de consommation électrique. Les grilles de calcul permettent quant à elles de passer à une échelle supérieure en terme de ressources disponibles. En effet, plusieurs entités se réunissent et partagent leurs ressources. Bien que cette architecture permette d'augmenter les ressources disponibles, les coûts sont l'hétérogénéité introduite par la diversité des machines entre chaque site et même entre les clusters d'un même site, et le caractère volatil des machines. À ces inconvénients viennent s'ajouter la disponibilité variable des machines ainsi que la charge du réseau, provenant des divers utilisateurs de la plateforme. Enfin, concernant les plateformes de calcul volontaire et les « desktop grids », leur principal avantage est le coût de la plateforme, qui est quasi nul. Mais le revers de cet avantage est l'obtention de moins bonnes performances aussi bien au niveau de la puissance de calcul que du réseau. Le fait est que les machines utilisées ne sont pas dédiées au calcul. Cependant, vu les caractéristiques des machines de travail actuelles, les performances sont correctes.

Pour conclure, il existe ainsi de nombreuses plateformes de calcul, chacune ayant ses avantages et ses inconvénients. Il convient à un utilisateur désireux de lancer une application de calcul parallèle de savoir quelle plateforme convient le mieux à la bonne exécution de son application.

Environnements d'exécution

Dans la seconde partie du chapitre ont été présentés des environnements complets et des plateformes d'exécution. Des environnements sont dits complets, parce qu'ils possèdent deux parties : une bibliothèque de programmation pour créer des applications, et une partie plateforme d'exécution permettant d'exécuter ces applications. Les environnements complets présentés dans ce chapitre sont MPI, ProActive, Jace, JaceP2P et JaceP2P-V2. Les plateformes d'exécution, telles que XtremWeb et XtremWeb-CH, ne permettent pas de programmer des applications à proprement parler. Elles servent d'interface pour l'exécution d'applications sur des architectures distribuées, comme des plateformes de calcul volontaire.

En premier lieu a été présentée non pas une plateforme mais une interface de programmation. MPI est une norme, mise en œuvre par divers environnements. Cette norme est devenu un standard de fait, car elle est très largement utilisée dans le milieu du calcul scientifique distribué. Cette norme définit des fonctions facilitant la programmation d'applications parallèles, mais également la gestion de leur exécution.

Dans un second temps nous avons présenté l'environnement complet ProActive. Cet environnement possède un très large éventail de fonctionnalités. Ceci rend cet environnement très efficace et robuste pour exécuter des applications sur des architectures distribuées. Cependant, le fait qu'il soit très complet lui confère une certaine lourdeur et entraîne une certaine complexité de programmation. De plus, par défaut cet environnement est totalement dédié aux modèles synchrones. Nous avons développé une bibliothèque afin de permettre l'utilisation d'algorithmes asynchrones, tout en ajoutant de nouvelles fonctionnalités de communication – ceci complétant l'environnement.

Nous avons ensuite présenté les plateformes d'exécution XtremWeb et XtremWeb-CH. Ces plateformes sont principalement pensées et dédiées à l'exécution d'applications parallèles sur des architectures de calcul volontaire. XtremWeb-CH a l'avantage de pouvoir rassembler des machines pouvant se trouver derrière des pare-feux et des réseaux « natés[6] ». Ces plateformes sont bien adaptées aux applications dont les tâches sont indépendantes.

Enfin, nous avons présenté les environnements Jace, JaceP2P et JaceP2P-V2. Ces environnements sont dédiés au calcul itératif asynchrone. Chacun de ces environnements apporte des innovations et de nouvelles fonctions, tout en ajoutant des mécanismes de tolérance aux pannes, de plus en plus poussés, ainsi qu'en améliorant le passage à l'échelle. Le dernier environnement, JaceP2P-V2, est totalement tolérant aux pannes et n'a pas de limite théorique, concernant le nombre de machines connectées, aussi bien dans des environnements sûrs, comme des clusters, que dans des environnements volatils.

Nous avons ainsi présenté divers environnements et plateformes d'exécution d'applications parallèles dans des environnements distribués. Chacun de ces environnements est plus ou moins dédié à un type d'application et/ou d'architecture d'exécution. Tous possèdent des avantages et des inconvénients, et il convient au programmeur de choisir l'environnement qui correspond au mieux à son application et à la plateforme d'exécution.

Ainsi, la majeure partie des travaux présentés dans cette thèse se concentre sur les applications basées le modèle itératif asynchrone, dont l'exécution se déroule sur des architectures distribuées et hétérogènes, telles que les grilles de calcul ou des clusters distribués. L'environnement utilisé pour la réalisation de ces travaux est JaceP2P-V2.

6. Une machine dans un réseau « naté » n'est pas atteignable depuis l'extérieur.

Deuxième partie

Placement de tâches

Dans cette partie nous présentons le premier axe de recherche de cette thèse. Les contributions portent sur le placement des tâches des applications parallèles et distribuées sur des architectures distribuées. Nous nous intéressons ici plus particulièrement aux applications itératives asynchrones. Les architectures qui nous intéressent ici sont des architectures à large échelle, distribuées, hétérogènes et comportant des machines de calcul volatiles – telles que les grilles de calcul.

Dans de tels environnements d'exécution, un placement efficace des tâches de calcul est essentiel, afin d'obtenir les meilleurs temps d'exécution possibles. Le choix d'une stratégie de placement doit prendre en compte plusieurs critères comme les caractéristiques de la plateforme d'exécution (caractéristiques des machines et du réseau) et celles des applications. Une telle stratégie peut aussi assurer que les machines choisies auront certaines caractéristiques minimales, comme par exemple la taille de la mémoire vive. En effet, certaines applications travaillent sur des jeux de données conséquents, et il est plus efficace d'avoir la possibilité de tout charger en mémoire centrale.

Un autre enjeu, dû à la volatilité des machines dans de telles architectures, est la gestion des pannes. Si une machine devient indisponible alors qu'elle participe au calcul, ce problème doit être géré de manière efficace, afin de pénaliser au minimum l'application. La technique utilisée ici repose sur la méthode du « checkpoint/restart », dont le principe est de sauvegarder régulièrement l'état du calcul afin de pouvoir le redémarrer en cas de panne d'une machine, sans perdre le travail de calcul déjà effectué. Cet aspect doit être pris en compte dans le placement des tâches, non seulement pour le placement initial, mais aussi pour le redéploiement d'une sauvegarde lors d'une panne.

Le chapitre 3 présente le contexte et l'état de l'art du placement de tâches de calcul sur des architectures distribuées. Ainsi, sont présentées les modélisations, aussi bien des architectures de calcul que des applications, et les classes d'algorithmes existants. Nous définissons également la problématique de nos travaux.

Le chapitre 4 présente trois algorithmes de placement, ainsi que leur intégration au sein de l'environnement JaceP2P-V2. Nous y présentons d'abord deux algorithmes issus de la littérature, que nous avons adaptés au modèle itératif asynchrone. Le troisième est un algorithme hybride et adaptatif, dédié au placement des tâches d'applications itératives asynchrones sur des architectures hétérogènes et volatiles. Cet algorithme est notre contribution majeure. Enfin, nous présentons une bibliothèque permettant la mise en œuvre et l'utilisation de tels algorithmes.

Le chapitre 5 présente les expérimentations que nous effectuées afin d'observer les performances de nos trois algorithmes de placement. Ces expérimentations nous ont permis de déterminer les gains apportés par ces algorithmes sur le temps d'exécution de deux applications itératives asynchrones dans des environnements hétérogènes et volatils.

Chapitre 3

Contexte et problématique

Ce chapitre a pour objectif de présenter le contexte et la problématique du placement de tâches d'applications parallèles sur des architectures distribuées et volatiles.

Dans un premier temps nous définissons notre problématique, constituant la motivation de nos travaux. Dans un second temps nous présentons les modélisations des architectures et des applications, ainsi que les définitions et notations qui seront utilisées dans la suite de ce document. Ensuite nous présentons deux classes d'algorithmes de placement offrant des solutions à ce problème. Dans la dernière partie est présentée la formalisation de notre problème. Celle-ci se décompose en deux parties, qui sont le placement initial des tâches et le remplacement de machines lors de la détection de pannes.

3.1 Problématique

Comme nous l'avons vu dans la partie précédente, le contexte de nos travaux est celui de l'exécution d'applications itératives asynchrones sur des architectures de type clusters distribués. Cependant, le problème du placement des tâches de ces applications, représenté par la figure 3.1, n'avait pas encore été étudié.

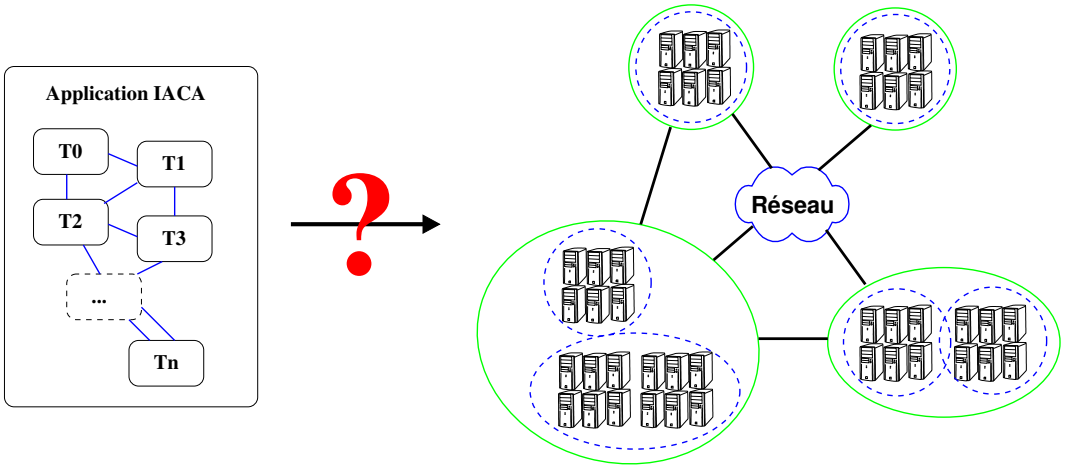

FIGURE 3.1 – Problématique du placement des tâches des applications IACA

En effet, comme nous l'avons vu dans la section 1.3.2, l'utilisation de méthodes itératives asynchrones permet d'obtenir de meilleures performances lorsque la plateforme d'exécution est une architecture distribuée, avec des liens de communication à forte latence et des machines de calcul volatiles. Cependant, la question qui est posée est la suivante : ces performances peuvent-elles être améliorées si le placement des tâches de calcul est choisi de manière efficace ? Dans l'affirmative, plusieurs critères peuvent être choisis pour orienter la sélection des machines. En effet, de par sa nature non prévisible, le modèle itératif asynchrone ne permet pas de définir un critère comme étant plus important que les autres. Ainsi les choix de la stratégie de placement peuvent être basés sur : privilégier la puissance des machines, ou privilégier la localité des tâches, ou privilégier le nombre de machines disponibles dans un cluster, ou encore la performance des liens de communication ...

Afin de répondre à cette question et en prémisses de nos travaux, nous avons d'abord réalisé une série d'expérimentations visant à vérifier la pertinence d'un placement optimisé des tâches sur les machines de calcul – le modèle itératif asynchrone étant particulier, une telle vérification était obligatoire.

Pour ce faire, nous avons mené une campagne d'expériences en utilisant une application typique du modèle itératif asynchrone. Cette application est une variante du calcul du gradient conjugué, décrite plus en détails dans la section 5.1.3. Nous avons utilisé deux tailles de problèmes, en faisant varier la largeur de bande de la matrice, afin de suivre son comportement. L'application était découpée en 64 tâches de calcul. La plateforme d'exécution que nous avons utilisée est Grid'5000, dont la description est donnée en section 5.1.1. L'architecture était composée de 102 machines, représentant un total de 320 cœurs de calcul, réparties en 5 clusters sur 5 sites. Le fait d'utiliser plusieurs sites permet d'introduire une certaine latence dans les communications entre les tâches. Pour l'exécution d'application, nous avons utilisé l'environnement JaceP2P-V2. Comme nous l'avons mentionné en section 2.2.4, cet environnement ne fournit aucune stratégie de placement des tâches. Les tâches sont affectées aux machines suivant leur ordre de connexion à la plateforme.

Afin de vérifier la pertinence d'un placement « intelligent » des tâches, nous avons élaboré un algorithme de placement simple. Le principe de cet algorithme consiste à classer dans un premier les clusters. Ici, les clusters sont triés suivant le nombre de machines qui les composent, en partant de celui en contenant le plus vers celui en contenant le moins. Ensuite les tâches sont placées, dans l'ordre de leur identifiant sur les machines. Chaque cluster est intégralement utilisé jusqu'à ce que toutes les tâches soient affectées à une machine.

Les résultats obtenus sont présentés dans le tableau 3.1, indiquant les temps d'exécution de l'application pour chaque taille de problème utilisée, avec et sans utilisation de l'algorithme de placement.

Taille du problème	Largeur de bande	Sans placement	Avec placement	Gain
550,000	350,000	129 s	97 s	25 %
550,000	totale	135 s	102 s	24 %
5,000,000	350,000	141 s	81 s	42 %
5,000,000	totale	149 s	88 s	41 %

TABLE 3.1 – Résultats de l'expérimentation pour vérifier la pertinence de l'utilisation d'un algorithme de placement

Dans un premier temps, ce que montre clairement ces résultats est le fait qu'un algorithme de placement, même aussi simple que celui utilisé ici, permet de réduire significativement le temps d'exécution d'une application itérative asynchrone. On remarque aussi que plus la taille du problème augmente, plus

un placement efficace devient important. Ainsi, les gains apportés par un placement efficace passent à l'échelle, en suivant la taille des données.

Il faut toutefois nuancer ces résultats et ne pas les généraliser pour toutes les applications itératives asynchrones. En effet, ce modèle étant très imprévisible, le comportement peut varier d'une application à une autre. Mais il est évident que le fait d'effectuer un placement recherché des tâches sur les machines de calcul ne peut qu'apporter des bénéfices sur le temps d'exécution des applications.

3.2 Définitions et modélisations

Cette section a pour objectif de présenter les diverses notations et choix effectués pour réaliser le placement de tâches. Dans un premier temps nous présentons la modélisation des plateformes d'exécution, que sont les architectures distribuées et volatiles. Dans un second temps la modélisation des applications itératives asynchrones est présentée. Enfin, la dernière section présente les spécificités du modèle itératif asynchrone, qui imposent certaines contraintes pour la réalisation du placement des tâches.

3.2.1 Modélisation des plateformes d'exécution

Les plateformes étudiées ici sont des architectures distribuées et volatiles, possédant de nombreuses machines, réparties en clusters, eux-mêmes pouvant être éloignés géographiquement. Nous présentons ici la modélisation générale de ces plateformes, en détaillant un paramètre important, qui est le degré d'hétérogénéité.

Modèle général

Afin d'effectuer un placement efficace des tâches de calcul, il faut connaître en détails les éléments constituants de la plateforme d'exécution. Les plateformes qui nous intéressent plus particulièrement sont hétérogènes et volatiles, comme par exemple des grilles de calcul ou des clusters distribués. La figure 3.2 présente une telle architecture.

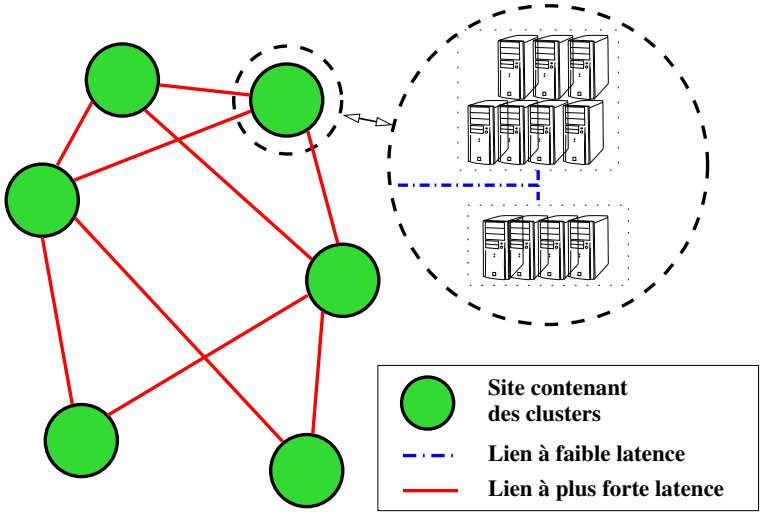

FIGURE 3.2 – Exemple d'une architecture distribuée et volatile

Comme le montre la figure, une telle architecture peut donc contenir plusieurs clusters, répartis sur plusieurs sites. Généralement, les clusters, et par extension les machines au sein d'un même site, sont reliés entre eux par des réseaux performants à faible latence. Les sites sont reliés entre eux par des liens un peu moins performants à plus forte latence. De même, tous les sites ne sont pas reliés directement les uns aux autres. Dans ce cas, un ou plusieurs sites relais sont utilisés, augmentant par là même la latence.

Ce type d'architecture peut être modélisé par un graphe à trois niveaux. Ceci est possible car les architectures visées remplissent les critères suivants :

- toutes les machines de calcul se trouvent sur un même niveau du graphe ;
- quand des machines communiquent entre elles en utilisant un réseau de même latence, alors celles-ci peuvent être rassemblées au sein d'un même cluster ;
- tous les clusters composant l'architecture, ainsi que respectivement les machines de calcul de ceux-ci, sont situés au même niveau du graphe et peuvent communiquer au travers du réseau de communication global de l'architecture.

Les trois niveaux de ce graphe sont :

- l'architecture – dans le cas de nos travaux la plateforme Grid'5000 ;
- les clusters ;
- les machines de calcul.

Considérons le graphe non orienté $GM(M, L)$ représentant une architecture de clusters distribués, avec $M = M_1, M_2...M_n$ l'ensemble des $|M|$ sommets, et L l'ensemble des $|L|$ arcs non dirigés reliant les sommets. Les sommets représentent les machines de calcul et les arcs les liens de communication entre elles. Un arc $L_{xy} \in L$ est une paire non ordonnée $(M_x, M_y) \in M$, représentant un lien de communication entre les machines de calcul M_x et M_y. Considérons $|M|$ comme étant le nombre de machines de calcul de l'architecture. Nous définissons aussi C comme étant une partition de GM, représentant l'ensemble des clusters, avec $|C|$ le nombre de clusters de l'architecture.

Sur ce graphe sont données les informations suivantes :

- une fonction $PM : M \rightarrow \mathbb{R}^+$ donne la puissance de calcul d'une machine ;
- une fonction $LL : L \rightarrow \mathbb{R}^+$ donne la latence d'un lien de communication ;
- une fonction $NM : C \rightarrow \mathbb{N}^+$ donne le nombre de machines de calcul contenues dans un cluster ;
- une fonction $ND : C \rightarrow \mathbb{N}^+$ donne le nombre de machines de calcul disponibles au sein d'un cluster – c'est-à-dire qui ne sont pas impliquées dans un calcul.

Dans la suite de ce document, nous utilisons les notations suivantes :

- $PM(M_i) = pm_i$ (puissance machine en MFlop) ;
- $LL(M_i, M_j) = ll_{ij}$ (latence d'un lien en milisecondes) ;
- $NM(C_i) = nm_i$ (nombre de machines du cluster C_i) ;
- $ND(C_i) = nd_i$ (nombre de machines disponibles dans le cluster C_i).

Nous pouvons alors définir les propriétés suivantes sur l'architecture :

- $P_{C_i} = \sum_{j=1}^{nm_i} pm_j$ comme étant la puissance totale du cluster C_i (puissance totale des machines du cluster) ;
- $P_{Cd_i} = \sum_{j=1}^{nd_i} pm_j$ comme étant la puissance totale des machines disponibles du cluster C_i (puissance des machines disponibles) ;
- $\overline{P_{C_i}} = \frac{P_{C_i}}{nm_i}$ comme étant la puissance moyenne d'un cluster C_i (puissance moyenne des machines du cluster) ;

– $\overline{P_{Cd_i}} = \frac{P_{Cd_i}}{nd_i}$ comme étant la puissance moyenne des machines disponibles d'un cluster C_i (puissance moyenne des machines disponibles du cluster).

Toutes ces propriétés sont essentielles pour un placement efficace des tâches sur une telle architecture. Concernant la partie liens de communication, deux paramètres peuvent être pris en compte : la latence et le débit. Dans des environnements distribués et volatils, la mesure de chacun d'eux est délicate et doit être permanente. C'est pourquoi dans notre étude nous avons décidé de n'utiliser que la latence, bien qu'il serait également intéressant d'utiliser le débit, voire une combinaison des deux.

Il existe aussi une autre donnée importante à prendre en compte, qui est le degré d'hétérogénéité de la plateforme. Nous proposons de quantifier cette donnée de la façon décrite ci-après.

Degré d'hétérogénéité

Dans un environnement volatil, les machines se connectent et se déconnectent, modifiant ainsi la structure même de l'architecture, entraînant également des changement dans les caractéristiques des machines de calcul. Lors du placement initial, la plateforme d'exécution présente une certaine hétérogénéité, qui peut évoluer au cours de l'exécution d'une application. Ce degré d'hétérogénéité, noté hd, est calculé en utilisant la formule de l'écart type relatif :

$$hd = \frac{\sigma_{PM}}{avg_{PM}} \tag{3.1}$$

dans laquelle avg_{PM} représente la puissance moyenne de calcul des machines, calculée suivant :

$$avg_{PM} = \frac{1}{|M|} \times \sum_{i=1}^{|M|} pm_i,$$

et σ_{PM} représente l'écart type portant sur la puissance de calcul des machines, calculée suivant :

$$\sigma_{PM} = \sqrt{\frac{1}{|M|} \times \sum_{i=1}^{|M|} (pm_i - avg_{PM})^2}.$$

Cette mesure du degré d'hétérogénéité de la plateforme d'exécution permet d'obtenir son cœfficient de variation, en terme de puissance de calcul. Ici, ne sont considérées que les valeurs réelles telles que $0 \leq hd \leq 1$, 0 dénotant une architecture très homogène, et 1 dénotant une architecture très hétérogène. Il est possible que hd soit supérieur à 1, indiquant alors un écart de puissance très important entre la machine la moins puissante et celle la plus puissante. Un écart supérieur à 1 permet simplement de quantifier cet écart, alors que ce qui nous intéresse ici est de déterminer l'homogénéité ou l'hétérogénéité de la plateforme d'exécution. Ainsi, lorsque la valeur fournie par l'équation (3.1) est supérieure à 1, celle-ci est ramenée à 1 – ainsi si $hd > 1$ alors $hd = 1$.

3.2.2 Modélisation des applications

Après avoir modélisé les plateformes d'exécution, nous présentons celle des applications. Tout comme pour les plateformes d'exécution, les applications peuvent être modélisées à l'aide de graphes. Ces graphes doivent modéliser aussi bien les tâches en elles-mêmes, que les communications, ou les dépendances qui existent entre elles.

Modèle général

Un grand nombre d'algorithmes d'ordonnancement représentent une application parallèle sous la forme d'un graphe, appelé DAG [70, 62, 45, 65], pour « Direct Acyclic Graph », ou graphe orienté sans cycle. La figure 3.3 représente un tel graphe.

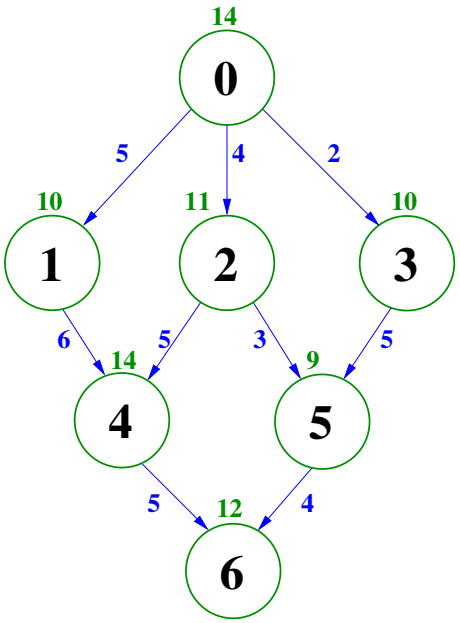

FIGURE 3.3 – Exemple d'un DAG

Comme on peut le voir sur la figure 3.3, chaque tâche, représentée par un cercle, possède un identifiant et est reliée aux autres par des liens orientés. Le chiffre situé en haut de la tâche indique son coût de calcul, et les chiffres situés près des liens correspondent au coût de communication lors d'échanges de données entre les tâches. Le fait que deux tâches soient reliées indique qu'il existe une dépendance entre elles – celle au bout de la flèche dépend de celle se trouvant à sa base. Cette dépendance est ici une précédence. Par exemple, la tâche 4 ne peut s'exécuter avant que les tâches 1 et 2 ne soient terminées et lui aient envoyé leurs résultats. Cette modélisation convient bien aux algorithmes synchrones (en utilisant un DAG pour modéliser une itération). Ceci est possible car à chaque itération, les tâches se synchronisent, envoient leurs données, et ne commencent l'itération suivante que lorsqu'elles ont reçu des mises à jour de toutes leurs dépendances. Une application itérative synchrone ne serait donc qu'une répétition de ce DAG.

Cependant, le sujet d'étude de cette partie est le placement des tâches des algorithmes itératifs asynchrones. Comme nous l'avons vu en section 1.3.2, il n'y a aucune synchronisation entre les tâches, ceci impliquant que les tâches peuvent se trouver au même instant t dans des itérations différentes. De plus, il n'existe aucune précédence entre les tâches, dans le sens où les tâches commencent l'itération suivante sans attendre de recevoir des mises à jour de leurs dépendances. C'est pourquoi la modélisation d'un algorithme itératif asynchrone ne peut utiliser un DAG. Le modèle permettant de préserver la philosophie des algorithmes asynchrones est le TIG [48, 6]. Un TIG, pour « Task Interaction Graph », ou graphe d'interaction de tâches, permet de ne modéliser que les relations de dépendances existantes entre les tâches. La figure 3.4 représente un tel graphe.

Comme le montre la figure 3.4, les tâches, représentées par des cercles, sont reliées aux autres tâches par des liens. Ceux-ci indiquent seulement que ces tâches dépendent les unes des autres. Ici, les dépendances sont à considérer comme de simples échanges de messages, et non pas comme des

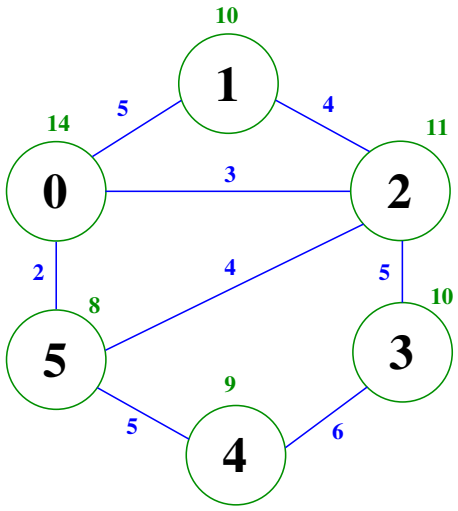

FIGURE 3.4 – Exemple d'un TIG

précédences. Les communications peuvent avoir lieu à n'importe quel moment durant l'exécution de la tâche. Il n'y a donc aucune dépendance temporelle.

Soit un TIG $GT(T, I)$, dans lequel $T = T_1, T_2, \ldots T_v$ représente l'ensemble des $|T|$ sommets, et $I \subset T \times T$ l'ensemble des arcs non orientés entre les tâches. Ici, les sommets T_i représentent les tâches de calcul et les arcs $I(T_i, T_j)$ les interactions entre les tâches. Sur ce graphe sont données les informations suivantes sur l'application :

- une fonction $CT : T \rightarrow \mathbb{R}^+$ donne le coût de calcul d'une tâche (en nombre d'opérations) ;
- une fonction $CI : I \rightarrow \mathbb{R}^+$ donne le coût de communication qu'il existe en deux tâches (en taille de données) ;
- une fonction $DT : T \rightarrow \mathbb{N}^*$ donne le nombre de dépendances que possède une tâche ;
- une fonction $DEP : T \rightarrow T$ donne la liste des dépendances d'une tâche.

Dans la suite de ce document, nous utilisons les notations suivantes :

- $CT(T_i) = ct_i$;
- $CI(T_i, T_j) = ci_{ij}$;
- $DT(T_i) = dt_i$;
- $DEP(T_i) = dep_i$.

Pour illustrer ces notations on peut noter sur la figure 3.4 que $ct_0 = 14$, $ci_{0,1} = 5$, et $dt_2 = 4$.

Spécificités du modèle itératif asynchrone et des environnements volatils

Comme énoncé précédemment, le modèle du TIG correspond bien à la modélisation des applications itératives asynchrones. Cependant, il existe certaines contraintes empêchant l'utilisation des algorithmes existants, du moins sans quelques modifications.

De nombreux algorithmes de placement proposent comme solution de placer plusieurs tâches sur une même machine de calcul, afin de réduire les coûts de communication. Cependant, comme les architectures d'exécution visées ont un caractère volatil, nous n'autorisons pas l'exécution de plusieurs tâches sur une même machine de calcul. La raison évidente est que placer plusieurs tâches sur une même machine entraîne une perte de performances pendant la phase de calcul, mais surtout lors du redéploiement des sauvegardes à la suite d'une panne d'une machine. En effet, lorsqu'une machine,

exécutant plusieurs tâches, tombe en panne, celle la remplaçant doit récupérer les dernières sauvegardes de toutes les tâches et les déployer. Les opérations de récupération des multiples sauvegardes sur les voisins peut entraîner une forte occupation du réseau, pénalisant ainsi les communications des autres tâches continuant leurs calculs. De même, des lenteurs peuvent apparaître lors du redémarrage des diverses tâches – chaque tâche démarrera plus lentement que si elle était seule à démarrer. Toutes ces opérations entraînent donc des pertes de performances, aussi bien au niveau de la puissance de calcul que de la bande passante du réseau, entraînant donc des pertes de temps. Cependant, du fait qu'aujourd'hui beaucoup de machines de calcul possèdent plusieurs cœurs de calcul, il est préférable d'utiliser une parallélisation de la tâche en elle-même, en utilisant par exemple des solveurs proposant une parallélisation interne adaptée aux machines multi-cœurs.

Un autre point important à prendre en considération concernant le modèle itératif asynchrone est la localité des tâches. En effet, avec ce modèle, plus une tâche reçoit fréquemment et rapidement ses données de mises à jour, ou dépendances, plus elle convergera rapidement. C'est pourquoi le fait d'essayer de préserver au maximum la localité des tâches peut avoir des bénéfices doubles : des communications et une convergence plus rapides.

Enfin, concernant la modélisation des tâches en elles-mêmes, il est difficile de pouvoir donner un coût de calcul. En effet, pour une tâche donnée, la charge peut changer entre chaque itération. Ceci provient du fait qu'une tâche peut ne pas recevoir de données de mise à jour, inhibant ainsi une partie du calcul. De même, pour certaines applications, comme par exemple celle de la résolution de l'équation d'advection/diffusion (décrite plus en détails dans le chapitre 5), la charge évolue en cours d'exécution. Ceci est un problème qui se situe en dehors de cette thèse. Ici, nous considérons les charges de calcul comme étant quasiment équivalentes, à un delta près. Ici la charge est prédéfinie en avance en donnant une estimation du nombre d'opérations qui seront effectuées. Un dernier point très important à considérer est le fait que le nombre d'itérations d'une tâche est impossible à prévoir par avance. Ainsi, nous ne pouvons estimer que le coût d'une seule itération et non le coût réel d'une tâche, qui serait le coût d'une itération multiplié par le nombre d'itérations.

3.3 Classes d'algorithmes

Dans la littérature, il existe de nombreux algorithmes de placement exploitant des graphes de type TIG. Ces algorithmes peuvent être classés en deux grandes catégories : ceux cherchant à minimiser la distance entre deux tâches dépendantes – le but étant de conserver un maximum de tâches interdépendantes au sein d'un même cluster –, et ceux cherchant à minimiser le temps d'exécution total de l'application. Dans cette section sont présentées successivement ces deux classes d'algorithmes.

3.3.1 Minimisation des liens externes

Les algorithmes de cette première classe, appelée « edge-cuts minimization » dans la littérature, ont pour objectif de minimiser l'utilisation des liens externes à un cluster. Leur but est de favoriser les échanges à l'aide d'un réseau à faible latence (en interne du cluster), plutôt que d'emprunter des liens à plus forte latence (ceux existants entre les sites). Ainsi leur objectif vise à placer les tâches pour qu'elles soient plus proches les unes des autres, afin d'optimiser l'échange des données entre elles.

Ici, ce sont les liens et leur coût qui sont le centre d'intérêt de ces algorithmes. À titre d'exemples, on peut citer ici Ghafoor et al. [21], ou Metis [42], Chaco [38], et PaGrid [41] comme des bibliothèques fournissant de tels algorithmes. Leur principal inconvénient est le fait qu'ils ne tiennent pas compte de la puissance des machines, et par extension, de l'hétérogénéité de la plateforme d'exécution. Ils ne se concentrent que sur la réduction de l'utilisation des liens de communication à forte latence. Ceci

montre que ces algorithmes privilégieront toujours les clusters avec un nombre plus élevé de machines de calcul, même si un autre possède seulement quelques machines de moins mais que celles-ci sont beaucoup plus puissantes.

3.3.2 Minimisation du temps d'exécution

Les algorithmes de cette seconde classe, appelée « execution time minimization », ont pour objectif de réduire le temps d'exécution total de l'application, dans sa globalité. Le but ici est d'optimiser le placements des tâches, afin que toutes les tâches terminent au plus tôt. Ceci revient souvent à ce que la tâche la plus lente (celle qui par exemple est placée sur la machine la plus lente, ou celle possédant la plus grosse charge de calcul, ou une combinaison des deux scénarii) termine au plus tôt.

Ici, ce sont les tâches et leur coût qui sont le centre d'intérêt de ces algorithmes. À titre d'exemples, on peut citer FastMap [61] et MiniMax [44]. Il est à noter qu'il existe un autre algorithme performant, qui est QM [55, 54] pour « Quick-Quality Map ». Celui-ci cherche à trouver pour chaque tâche la machine de calcul lui permettant de finir au plus tôt. QM travaille au niveau des tâches alors que les autres algorithmes travaillent au niveau de l'application globale. Nous pouvons également citer ici l'algorithme PROPHET [68], qui permet à l'utilisateur de choisir entre plusieurs politiques de placement (en l'occurrence deux) en fonction de l'hétérogénéité du réseau de communication. Alors que QM présente de bonnes performances par rapport aux autres algorithmes de placement connus, il n'existe pas, à notre connaissance, de comparaisons de performance entre PROPHET et les autres algorithmes de la littérature. De plus, cet algorithme se limite à trois topologies d'interconnexion entre les clusters.

Le principal inconvénient de ces algorithmes vient du fait que bien souvent ils placent plusieurs tâches sur une même machine, ceci afin d'optimiser les communications. Comme il a été précisé précédemment, pour des raisons de tolérance aux pannes, ce type d'arrangement n'est pas possible dans le cadre de notre étude.

3.4 Définition du problème

Comme le montrent les sections précédentes, il existe deux types de solutions permettant d'effectuer le placement des tâches sur une plateforme d'exécution distribuée et volatile. Cependant, nous avons aussi présenté les limitations de ces approches. Cette section a pour objectif de définir la problématique que nous avons traitée, qui se décompose en deux points. Le premier point concerne le placement initial des tâches de calcul, et le second point concerne le choix d'une machine de remplacement lorsqu'une machine tombe en panne.

3.4.1 Placement initial

Considérons une application parallèle IACA, notée App, dont la représentation sous la forme d'un TIG donne un graphe GT. Sur ce graphe le poids des nœuds représente le coût d'une itération et le poids des liens le coût d'un échange de dépendance. Cette application est exécutée sur une plateforme de calcul, représentée par un graphe, noté GM. Sur ce graphe la valeur des nœuds représente la puissance des machines et celle des liens représente la latence entre les machines. Le temps d'exécution de l'application App, noté $TE(App)$, est considéré comme étant le temps d'exécution de la tâche la plus lente – celle terminant au plus tard. En effet, ceci vient du fait qu'une application itérative asynchrone se termine quand toutes les tâches ont détecté la convergence et atteint l'approximation souhaitée par

l'utilisateur. C'est pourquoi le temps d'exécution de l'application dépend de la tâche terminant au plus tard. Pour définir le temps d'exécution de l'application, la formule suivante est utilisée :

$$TE(App) = \max_{i=1...v}(TE(T_i)) \tag{3.2}$$

dans laquelle, le temps d'exécution d'une tâche est donné par la formule :

$$TE(T_i) = nbIt_i \times \frac{ct_i}{pm_i} \tag{3.3}$$

où $nbIt_i$ représente le nombre d'itérations de la tâche T_i, ct_i sa charge de calcul et pm_i la puissance de la machine sur laquelle elle est placée. Dans cette formule les communications n'apparaissent pas car le modèle itératif asynchrone permet le recouvrement de celles-ci par du calcul. Or, comme nous l'avons déjà mentionné, il est impossible de prédire de manière exacte le nombre d'itérations qu'effectuera une tâche dans le modèle itératif asynchrone. Cependant, comme nous avons pu le remarquer, plus les messages de dépendances arrivent rapidement, plus le nombre d'itérations est faible. C'est pourquoi les communications doivent entrer en ligne de compte dans le calcul du temps d'exécution d'une tâche du modèle itératif asynchrone. Nous avons choisi d'évaluer le temps d'exécution d'une tâche de la façon suivante :

$$TE(T_i) = \frac{ct_i}{pm_i} + \sum_{j \in J} ci_{ij} \times ll_{ij}. \tag{3.4}$$

Dans cette formule, ct_i représente la charge de calcul de la tâche T_i, pm_i la puissance de calcul de la machine M_i sur laquelle est placée la tâche T_i ; J représente l'ensemble des dépendances de T_i, ci_{ij} la charge de communication entre les tâches T_i et T_j, et ll_{ij} la latence du lien de communication existant entre les machines sur lesquelles sont placées T_i et T_j. Comme le montre cette formule, le temps d'exécution d'une tâche dépend dans un premier temps naturellement de sa charge de calcul et de la puissance de la machine sur laquelle elle est placée, mais aussi de la charge de communication qu'elle a avec ses dépendances et de la latence des liens qui les relient entre elles. Pour rappel, nous ne considérons dans cette étude que la latence des liens et pas leur débit.

Il est important de noter ici que dans le modèle itératif asynchrone, il est impossible de prédire le nombre d'itérations que va effectuer une tâche. Il est donc difficile d'évaluer a priori la charge de calcul ct_i ; cette charge ct_i est donc estimée comme le coût d'une itération – bien qu'il ait été montré qu'estimer cette charge n'est pas chose facile. Pour rappel, nous considérons tous les ct_i comme étant équivalents, à un delta près.

Pour résumer, l'objectif du placement initial est de favoriser au mieux l'assignation des tâches aux machines de calcul, en tenant compte des paramètres suivants : l'hétérogénéité de la plateforme d'exécution et les caractéristiques des applications itératives asynchrones. Le but étant d'optimiser le temps d'exécution de l'application, pour qu'il soit le plus faible possible. Ainsi, nos travaux portent sur la minimisation de ce temps d'exécution, soit réaliser l'objectif suivant :

$$TE(App) = min(\max_{i=1...v}(TE(T_i))). \tag{3.5}$$

Il est à noter ici que ce problème est tout à fait similaire au problème de partitionnement de graphes et à l'assignation de tâches. Ce problème de placement de tâche est par conséquent NP-complet [34].

3.4.2 Remplacement de machines

Dans des environnements volatils, les machines de calcul peuvent se déconnecter à tout moment durant l'exécution d'une application. Comme décrit dans la section 2.2.4, la plateforme JaceP2P-V2 propose des mécanismes permettant la tolérance aux pannes, en utilisant la méthode des sauvegardes non coordonnées, afin de sauvegarder l'état des calculs. Lorsqu'une machine se déconnecte en cours de calcul, la plateforme doit la remplacer en choisissant la « meilleure » machine disponible afin de redémarrer la tâche à partir de la dernière sauvegarde effectuée.

Cette machine de remplacement doit être la « meilleure » au moment de la détection de la panne, suivant l'algorithme de placement utilisé. Celui-ci doit trouver cette remplaçante parmi les machines disponibles dans la plateforme d'exécution. Comme cette plateforme évolue au cours du temps, avec les connexions et déconnexions de machines de calcul, l'algorithme de placement doit garder en permanence une vue complète et à jour de la plateforme. Chaque évènement concernant la plateforme doit être traité le plus rapidement possible afin, par exemple, d'éviter de choisir une machine de remplacement qui vient tout juste de se déconnecter. De même, avec les changements au sein de la plateforme d'exécution, les critères de sélection des machines de calcul doivent évoluer, ceci afin de toujours choisir les meilleures machines – par exemple, une machine qui a été choisie lors du placement initial ne le serait plus en cours d'exécution, du fait de l'évolution de la plateforme.

Un autre problème se pose lorsqu'il y a eu de multiples pannes lors de l'exécution de l'application : certaines tâches ont pu migrer de machine en machine, voire de cluster en cluster, et donc le placement initial a totalement changé. En effet, avant l'exécution de l'application, l'algorithme choisit les meilleures associations entre les machines et les tâches, afin d'optimiser au mieux le temps d'exécution de l'application. Ce placement initial doit tenir compte non seulement de la puissance des machines, mais aussi de la proximité des tâches afin de ne pas utiliser des liens de communication pénalisants. C'est pourquoi, après avoir subi de nombreuses pannes et donc de nombreux redéploiements, certaines assignations peuvent ne plus satisfaire les critères de placements (la tâche n'est pas sur la machine la plus puissante, ou elle est trop loin de ses dépendances ...). De ce fait, un algorithme de placement doit évoluer dynamiquement en fonction de la plateforme d'exécution, qui est ici volatile.

3.5 Conclusion

Dans ce chapitre nous avons présenté le contexte et l'état de l'art concernant le placement des tâches d'applications itératives asynchrones sur des plateformes d'exécution distribuées et volatiles. Les différentes modélisations et définitions, aussi bien des architectures d'exécution que des applications, ont été données.

Ainsi une architecture est modélisée par un graphe à trois niveaux : les machines de calcul, les clusters, et l'architecture globale. Chaque composant est ainsi modélisé, en incluant les liens de communication. Pour la modélisation des applications, notre choix s'est porté sur l'utilisation d'un TIG représentant les applications sous la forme d'un graphe non orienté. Ce graphe ne présente pas de relations de précédences, mais seulement des relations de dépendances entre les tâches.

Dans une seconde partie, nous avons présenté les deux principales classes d'algorithmes de placement. La première, contenant les algorithmes d'optimisation des liens externes, a pour but de limiter l'utilisation des liens de communication pénalisants. Sont considérés comme pénalisants les liens à forte latence, comme ceux reliant différents sites. La seconde classe, celle des algorithmes visant à optimiser le temps d'exécution des tâches, a pour but de trouver pour chaque tâche la machine de calcul lui permettant de terminer au plus tôt. Ces deux classes d'algorithmes apportent chacune de bons points, mais en apportent d'autres en contradiction avec nos contraintes : comme la non prise en compte de

la puissance des machines de calcul pour la première classe, et le placement de plusieurs tâches sur une même machine pour la seconde classe. De plus, aucun de ces algorithmes ne propose, à notre connaissance, des mécanismes de gestion de la tolérance aux pannes et, en même temps, une prise en compte du degré d'hétérogénéité de la plateforme d'exécution, pouvant varier au cours de l'exécution de l'application.

Enfin, nous avons présenté notre problématique, qui se décompose en deux parties : le placement initial et les mécanismes de tolérance aux pannes. Pour le placement initial, nous avons vu que celui-ci doit tenir compte aussi bien des caractéristiques spécifiques des applications itératives asynchrones que de celles des plateformes d'exécution hétérogènes et volatiles. Pour les mécanismes de remplacement de machines, nous avons souligné l'importance du fait que l'algorithme de placement doit garder en temps réel une vue d'ensemble de la plateforme d'exécution, afin de choisir au mieux les machines de remplacement. De surcroît, l'algorithme doit toujours essayer de satisfaire au mieux les critères de placement, qui doivent évoluer en même temps que l'hétérogénéité de la plateforme d'exécution.

Chapitre 4

Algorithmes proposés

Après avoir présenté dans le chapitre précédent les modélisations et l'état de l'art du placement de tâches, ainsi que nos objectifs, ce chapitre présente nos contributions.

Dans un premier nous nous intéressons aux algorithmes de placement. Nous présentons tout d'abord deux algorithmes, issus de la littérature, que nous avons adaptés au modèle itératif asynchrone. Chacun des ces algorithmes représente une des deux classes présentées précédemment dans la section 3.4 : l'un pour les algorithmes cherchant à minimiser l'utilisation des liens externes (entre les clusters) qui peuvent être pénalisants, l'autre pour ceux visant à réduire de le temps d'exécution des tâches des applications. Le premier, FT-FEC, représente la première classe, et le second, FT-AIAC-QM, représente la seconde classe d'algorithmes de placement. Nous avons adapté chacun de ces algorithmes au modèle itératif asynchrone, et nous leur avons aussi ajouté des mécanismes permettant le remplacement de machines lorsqu'une panne survient lors de l'exécution d'une application – pour rappel, ces mécanismes ne sont pas prévus dans leurs versions originales. Ensuite nous présentons notre contribution majeure, MAHEVE, qui est un algorithme de placement hybride. Cet algorithme adapte sa politique de placement en fonction du degré d'hétérogénéité de la plateforme d'exécution. De plus, il intègre toutes les fonctionnalités nécessaires à la prise en compte non seulement des changements d'hétérogénéité de la plateforme, mais aussi de son caractère volatil en proposant des fonctions pour la tolérance aux pannes.

Dans un second temps, nous présentons une bibliothèque permettant la mise en œuvre d'algorithmes de placement, et s'intégrant à l'environnement JaceP2P-V2. Ceci permet à cet environnement de pouvoir bénéficier d'algorithmes permettant d'optimiser le placement des tâches de calcul, réduisant ainsi le temps d'exécution des applications.

4.1 Algorithmes de placement

Dans cette section sont présentés trois algorithmes de placement. Les deux premiers, FT-FEC et FT-AIAC-QM, que nous avons adaptés à l'exécution d'applications itératives asynchrones sur des architectures volatiles, sont issus de la littérature. Il est à noter ici que nous avons voulu préserver au maximum l'esprit de ces deux algorithmes lors de nos modifications. Nous présentons également une étude préliminaires montrant les performances de ces deux algorithmes.

Le troisième algorithme, MAHEVE, est totalement dédié à l'exécution d'applications itératives asynchrones sur des architectures volatiles. Il intègre dans sa conception la gestion de la variation du degré d'hétérogénéité de la plateforme d'exécution ainsi que des mécanismes de tolérance aux pannes.

4.1.1 FT-FEC

Ce premier algorithme, FT-FEC pour « Fault Tolerant Farhat EdgeCuts », est un algorithme de la classe visant à réduire l'utilisation de liens de communication pénalisants entre les tâches. Il est basé sur l'algorithme de Farhat [29].

Il a été choisi pour ses caractéristiques de partitionnement particulières. Les algorithmes classiques de partitionnement classiques tels que Metis [42] et Chaco [38] sont appelés « algorithmes de partitionnement multi-niveaux » – ces algorithmes peuvent être utilisés pour le partitionnement de graphes, comme par exemple dans [47]. Leur principal inconvénient est qu'ils ne permettent pas de tenir compte de l'hétérogénéité des plateformes d'exécution que nous visons. En effet, ces algorithmes considèrent que les machines de calcul et les liens de communication sont homogènes. Ces algorithmes reposent sur des méthodes d'expansion de graphe (« graph growing » dans la littérature), notées GGP, ou des méthodes d'expansion gloutonne de graphe (« greedy graph growing » dans la littérature), notées GGGP. Ces méthodes effectuent une division du graphe de tâches, créant ainsi des partitions. Ces partitions sont soit au nombre de deux, soit en un nombre d'une puissance de deux (2, 4, 8, 16...). Pour le cas des applications que nous utilisons, nous ne pouvons pas déterminer à l'avance le nombre de partitions qui seront nécessaires. En effet, ce nombre dépend essentiellement du nombre de machines disponibles et connectées à l'architecture d'exécution ainsi que du nombre de tâches de l'application.

Notre choix s'est porté sur l'algorithme de Farhat car il présente l'avantage de permettre le partitionnement d'un graphe en un nombre arbitraire de partitions – et non plus en un nombre puissance de deux – en évitant ainsi une bissection récursive. L'algorithme 3 donne le pseudo-code de FT-FEC.

L'objectif de cet algorithme est de minimiser l'utilisation des liens entre les clusters. Pour ce faire, il réalise des regroupements de tâches – ce principe porte le nom de « clusterisation » des tâches. Comme nous n'autorisons qu'une seule tâche par machine de calcul, l'algorithme vérifie dans un premier temps que cette condition est validée (ligne 1). Dans le cas contraire, l'algorithme se termine. Sinon, les clusters sont triés par ordre décroissant, suivant leur nombre de machines disponibles (ligne 4). Ensuite les tâches sont triées (ligne 5), également par ordre décroissant suivant leur nombre de dépendances. Ceci permet de donner une priorité plus forte aux tâches placées en tête de liste – ceci assurant que les tâches ayant le plus de dépendances devraient être placées avec un maximum de celles-ci sur le cluster possédant le plus grand nombre de machines disponibles.

Ensuite, l'algorithme entre dans une boucle (ligne 8), dont il ne sort que lorsque toutes les tâches ont été placées. Dans cette partie, il essaie de placer un maximum de tâches au sein d'un cluster. Les clusters sont choisis dans leur ordre de tri (ligne 10), et les variables associées sont initialisées (lignes 16 à 18). Concernant les tâches de calcul, une liste temporaire *listeTemporaire* est utilisée, celle-ci contenant les tâches en cours de traitement. Cette liste est remplie avec une nouvelle tâche lorsque la liste est vide (lignes 20 à 22), ou alors complétée avec des dépendances d'une tâche nouvellement placée (ligne 28). Afin de placer une tâche dans un cluster, il évalue (ligne 24) s'il reste assez de place dans le cluster pour placer la tâche courante ainsi qu'un nombre minimal de ses dépendances. Ce nombre est calculé en appliquant un coefficient δ (qui est un paramètre de l'algorithme défini entre 0 et 1) au nombre de dépendances de la tâche. Ici ne sont considérées dans ce nombre de dépendances que celles qui ne sont pas encore placées. Si le placement est possible, la tâche en cours de traitement est placée dans une liste *tâchesAssociées* (ligne 26), contenant les tâches qui seront associées au cluster en cours, et est supprimée des tâches à traiter (ligne 25). Les dépendances de cette tâche sont alors ajoutées dans la liste des tâches à traiter (ligne 28). L'ajout de ces tâches respecte le classement suivant le nombre de dépendances des tâches. Ce mécanisme permet de laisser la priorité aux tâches possédant le plus grand nombre de dépendances, et lorsqu'une tâche est placée, la priorité est donnée à ses dépendances, dans le but de favoriser les communications locales. Enfin, le nombre de places disponibles dans le cluster est décrémenté (ligne 27).

Algorithme 3 : Algorithme de FT-FEC

Entrées : *lc* (liste des clusters), *lt* (liste des tâches), δ (facteur de sélection de dépendances), *dim* (valeur de réduction de δ)

Sorties : *mp* (placement des tâches)

```
1  si le nombre de machines disponibles < nombre de tâches alors
2  |  placement impossible
3  fin
4  clustersTriés ← trier la liste des clusters lc
5  tâchesTriées ← trier la liste des tâches lt
6  changeCluster ← vrai
7  listeTemporaire ← vide
8  tant que toutes les tâches ne sont pas placées faire
9  |  si changeCluster alors
10 |  |  clusterCourant ← cluster suivant dans clustersTriés
11 |  |  si plus de cluster disponible alors
12 |  |  |  δ ← δ − dim
13 |  |  |  remise au début de la liste clustersTriés
14 |  |  |  clusterCourant ← cluster suivant dans clustersTriés
15 |  |  fin
16 |  |  nombrePlaces ← placesDisponibles(clusterCourant)
17 |  |  changeCluster ← faux
18 |  |  tâchesAssociées ← vide
19 |  fin
20 |  si aucune tâche dans listeTemporaire alors
21 |  |  ajouter la première tâche de tâchesTriées dans listeTemporaire
22 |  fin
23 |  tâcheCourante ← première tâche de listeTemporaire
24 |  si (1+nombreDep(tâcheCourante) × δ) <= nombrePlaces) alors
25 |  |  enlever tâcheCourante de listeTemporaire
26 |  |  ajouter tâcheCourante dans tâchesAssociées
27 |  |  nombrePlaces ← nombrePlaces − 1
28 |  |  ajouter dépendances(tâcheCourante) dans listeTemporaire
29 |  sinon
30 |  |  changeCluster ← vrai
31 |  |  associer tâchesAssociées avec clusterCourant
32 |  |  ajouter l'association dans mp
33 |  fin
34 fin
35 renvoyer mp
```

Lorsqu'il n'y a plus de place disponible dans le cluster (ligne 29), l'algorithme doit changer de cluster (ligne 30). Ceci implique l'enregistrement des associations entre les tâches et le cluster (ligne 31), la liste des tâches étant contenues dans *tâchesAssociées*. Cet enregistrement est effectué dans le placement final (ligne 32). Lorsqu'il n'y a plus de clusters disponibles (ligne 11) et qu'il reste des tâches à traiter, le paramètre δ est diminué de *dim* (ligne 12). Le paramètre *dim* permet d'influencer la rapidité de réduction de δ. Cette diminution de δ permet d'obtenir une plus grande souplesse lors de la vérification de la possibilité du placement d'une tâche sur un cluster (ligne 24). Ainsi, l'algorithme s'adapte à plateforme d'exécution et à l'application en relâchant certaines contraintes, et de ce fait une application peut toujours obtenir un placement de ses tâches sur une architecture donnée – du moment qu'il y ait au moins autant de machines de calcul que de tâches. Une fois que toutes les tâches sont affectées à un cluster, et que les associations résultantes sont enregistrées dans le placement final *mp*, ce dernier est retourné.

La complexité de cet algorithme est calculée de la façon suivante : le tri des clusters s'effectue en $O(|C|log|C|)$ (parcours de la liste des clusters et insertion dans une liste triée), le tri des tâches en $O(|T|log|T|)$ (parcours de la liste des tâches et insertion dans une liste triée), et la phase d'affectation des tâches aux clusters en $O(|T|(|C| + |T|))$ (parcours de toutes les tâches avec affection aux clusters en parcourant la liste des dépendances).

Comme son nom le mentionne, FT-FEC possède une fonction qui est utilisée pour la tolérance aux pannes. Ici, comme l'algorithme tente de remplir au maximum les clusters lors de l'élaboration du placement initial, le choix de la machine de remplacement est limité. La fonction essaye dans un premier temps de trouver une machine disponible dans le même cluster. Ceci est possible si quelques unes n'ont pas été utilisées (à cause de la vérification du nombre de places disponibles à la ligne 24), ou si des machines ont rejoint le cluster entre la création du placement initial et la détection de la panne. Le fait de choisir une machine dans le même cluster permet de préserver l'objectif de l'algorithme, en gardant une proximité des tâches interdépendantes. S'il n'y a pas de machine disponible dans le même cluster, la fonction effectue un nouveau tri des clusters (identique à celui de la ligne 4) afin de prendre en compte les changements intervenus sur la plateforme d'exécution. Ensuite, une machine du premier cluster de la liste *clustersTriés* est renvoyée.

4.1.2 FT-AIAC-QM

Le second algorithme que nous présentons est FT-AIAC-QM, pour « Fault Tolerant Asynchronous Iteration Asynchronous Quick-quality Map ». Cet algorithme de placement fait partie de la seconde classe d'algorithmes ; ceux visant à réduire le temps d'exécution des tâches. Il est basé sur l'algorithme QM [55, 54] et nous l'avons modifié afin qu'il corresponde à nos critères de placement. Cet algorithme cherche avant tout à favoriser la puissance brute des machines, c'est-à-dire leur puissance de calcul. L'algorithme original ne tient pas compte des communications, du moins pas de manière explicite. En plus de l'avoir modifié, nous lui avons ajouté une fonction permettant la sélection d'une machine de remplacement lorsqu'une panne est détectée en cours de calcul.

FT-AIAC-QM est un algorithme destiné à résoudre le problème du placement des tâches d'une application itérative asynchrone dans des environnements volatils. L'idée principale de cet algorithme est d'effectuer un premier placement, qui est ensuite raffiné par itérations successives. Au cours des itérations de raffinement, chaque tâche peut migrer vers une meilleure machine : une meilleure machine est une machine qui procure à la tâche un temps d'exécution plus petit que celui donné par celle où elle est actuellement placée. Toutes les machines ne sont pas considérées durant chaque itération, grâce à un facteur de recherche, noté f avec $0 \leq f \leq 1$. L'algorithme 4 donne le pseudo code de FT-AIAC-QM.

Dans un premier temps, toutes les machines sont classées par ordre décroissant en fonction de leur puissance pm_i (ligne 1), et chaque tâche est placée sur une machine (ligne 2). Ici les tâches sont choisies

Algorithme 4 : Algorithme de FT-AIAC-QM

Entrées : lm (liste des machines), lt (liste des tâches), f (facteur de recherche)
Sorties : mp (placement des tâches)

```
1  trier les machines
2  placer les tâches sur les machines
3  marquer toutes les tâches comme libre
4  r ← 1
5  tant que une tâche est libre faire
6      pour chaque tâche T_i avec T_i libre faire
7          m_{T_i} ← machine sur laquelle est placée T_i
8          m_m ← m_{T_i}
9          pour k = 0; k < f·|M|/r; k++ faire
10             sélectionner une machine m_h au hasard dans [0, |M|/r]
11             si TE(T_i,m_h) < TE(T_i,m_m) alors
12                 m_m ← m_h
13             fin
14         fin
15         pour chaque machine m_v au voisinage de dep_i faire
16             si TE(T_i,m_v) < TE(T_i,m_m) alors
17                 m_m ← m_v
18             fin
19         fin
20         si m_m ≠ m_{T_i} alors
21             placer T_i sur m_m – soit m_{T_i} ← m_m
22             mettre à jour TE(T_i)
23             mettre à jour TE(dep_i)
24         fin
25         marquer T_i comme fixe
26     fin
27     r ← r + 1
28 fin
29 pour chaque tâche T_i faire
30     ajouter dans mp l'association (T_i, m_{T_i})
31 fin
32 renvoyer mp
```

dans l'ordre de leur identifiant, c'est-à-dire que T_0 est placée sur la première machine du classement, T_1 sur la seconde, et ainsi de suite. Toutes les tâches sont marquées comme étant *libres* (ligne 3). Ce marquage indique que la tâche peut changer de machine d'affectation. L'algorithme rentre alors dans une boucle, dont il ne sortira que lorsque toutes les tâches seront marquées comme étant *fixes* (ligne 5). Une tâche est dite *fixe* quand elle est associée à une machine et que son temps d'exécution $TE(T_i)$ n'est plus sensé changer. Dans un premier temps, l'algorithme va rechercher une meilleure machine pour chaque tâche T_i *libre*. Il garde la trace du *nombre de tours* r de recherche avec $(r > 0)$. Cette variable permet de réduire à chaque tour effectué le nombre de machines à considérer. Avant de commencer la recherche, la machine courante m_{T_i} sur laquelle est placée la tâche T_i est gardée en mémoire (ligne 7), et la meilleure machine m_m est initialisée avec la machine courante M_{T_i} (ligne 8).

L'algorithme va alors rechercher une meilleure machine m_m pour la tâche T_i (lignes 9 à 14). Pour ce faire, il va choisir au hasard $\frac{f \cdot |M|}{r}$ machines, notées m_h, sélectionnées dans l'intervalle $[0, \frac{|M|}{r}[$ (lignes 9 et 10). Ici le facteur de recherche $f \in]0, 1]$ à l'aide de la variable r contrôlent la proportion des machines qui vont être considérées. Ainsi plus r est grand (plus il y a eu de tours de recherche) moins il y a de machines à prendre en compte. À l'inverse, plus le paramètre f est grand, soit proche de 1, plus le nombre de machines considérées sera élevé. Ce paramètre permet de régler la finesse de recherche de l'algorithme, et par conséquent sa vitesse de convergence. Si le temps d'exécution de la tâche T_i sur la machine m_h, noté $TE(T_i, m_h)$, est inférieur à son temps d'exécution sur sa meilleure machine actuelle m_m, noté $TE(T_i, m_m)$ (ligne 11), alors la meilleure machine de T_i devient m_h (ligne 12). Le fait d'utiliser une part de hasard introduit de l'indéterminisme. Cependant, celui-ci est faussé de par le fait que les machines sont triées selon leur puissance et que les tâches sont traitées dans le même ordre. On pourrait alors se demander si l'utilisation d'une solution plus directe ne serait pas à envisager ici.

Ensuite intervient une autre recherche plus fine prenant en compte la localité des tâches (lignes 15 à 18). Dans l'algorithme original, les machines sélectionnées ici sont celles sur lesquelles sont placées les dépendances de T_i, pour rappel notées dep_i. Placer plusieurs tâches sur une même machine de calcul permet de réduire les coûts de communication, mais induit de lourds inconvénients pour la tolérance aux pannes. Or, comme énoncé au chapitre précédent, nous n'autorisons pas l'exécution de plusieurs tâches sur une même machine de calcul. C'est pourquoi nous avons modifié cette recherche. Maintenant, l'algorithme va sélectionner des machines m_v se trouvant dans le voisinage de celles sur lesquelles sont placées les dépendances de T_i (ligne 15). Sont considérées comme faisant partie du voisinage les machines dont la latence du lien de communication qui les relie est faible (inférieure à 10ms). Si le temps d'exécution de la tâche T_i sur la machine m_v, noté $TE(T_i, m_v)$, est inférieur à son temps d'exécution sur sa meilleure machine actuelle m_m, noté $TE(T_i, m_m)$ (ligne 16), alors la meilleure machine de T_i devient m_v (ligne 17).

Une fois les deux phases de recherche d'une meilleure machine terminées, l'algorithme vérifie si une telle machine a pu être trouvée (ligne 20). Si tel est le cas, la tâche T_i est donc associée à la machine m_m (ligne 21) et son temps d'exécution $TE(T_i)$ est mis à jour. Ensuite les temps d'exécution de toutes les dépendances de T_i, notés $TE(dep_i)$ sont mis à jour (ligne 23). En effet, comme le temps d'exécution d'une tâche dépend en partie des communications qu'elle a avec ses dépendances, le fait de changer de place une tâche peut modifier ses temps de communication. Il est important de noter ici, que si, lors de cette phase de mise à jour, le temps d'exécution d'une tâche augmente, alors celle-ci est marquée comme *libre*, afin de pouvoir migrer à nouveau pour trouver un meilleur placement.

Enfin, que la recherche ait été fructueuse ou non, la tâche T_i est marquée comme *fixe* (ligne 25), indiquant ainsi que cette tâche a été traitée durant ce tour. Lorsque toutes les tâches ont été traitées au cours d'un même tour, la variable r est incrémentée (ligne 27). Lorsque plus aucune tâche n'est marquée comme *libre*, l'algorithme se termine et crée le placement final mp en lui ajoutant les associations des (T_i, m_{T_i}) (lignes 29 à 31), puis le renvoie (ligne 32).

La complexité de l'algorithme original est de $\Theta(f \cdot |M| \cdot |T| \cdot (ln(r) + 0.577))$ où $|T|$ est le nombre de

tâches et $|M|$ le nombre de machines. Les modifications que nous lui avons apportées, concernant les lignes 15 à 19, ajoutent un parcours des tâches, ajoutant ainsi un facteur $|T|$, impliquant une complexité finale de $\Theta(f \cdot |M| \cdot |T|^2 \cdot (ln(r) + 0.577))$.

Comme son nom l'indique, FT-AIAC-QM possède une fonction qui est utilisée pour la tolérance aux pannes. Le but de l'algorithme étant de minimiser le temps d'exécution des tâches en utilisant les machines les plus puissantes, le choix d'une machine de remplacement doit suivre la même direction. Cependant, dû à notre adaptation de l'algorithme au modèle itératif asynchrone, le choix doit aussi tenir compte de la localité des tâches, qui leur permet de converger plus rapidement. De ce fait, lorsqu'une machine tombe en panne, la fonction cherche en premier lieu si une machine est disponible dans le même cluster pour la remplacer. Si tel est le cas, alors cette dernière est choisie. Sinon, la fonction va rechercher la machine disponible la plus puissante présente dans l'architecture pour remplacer celle en panne.

4.1.3 Étude préliminaire

Dans cette section, nous allons nous intéresser aux comportements des deux algorithmes de placement décrits précédemment, FT-AIAC-QM et FT-FEC.

Cette étude préliminaire s'est déroulée en utilisant la plate-forme Grid'5000 et différentes architectures permettant de faire varier le degré d'hétérogénéité. Pour chaque architecture, 128 machines ont été utilisées, réparties sur 4 clusters de 3 sites pour chacune d'entre elles. L'application utilisée ici repose sur une version asynchrone du NAS [18] Kernel CG. Cette application est décrite dans le chapitre suivant. La taille de la matrice creuse utilisée est de 5000000 de côté et l'application a été découpée en 64 tâches. La largeur de bande utilisée dans la matrice définit 2 dépendances par tâche. L'environnement JaceP2P-V2 a été utilisé pour l'exécution de l'application. Les résultats de ces expérimentations sont donnés dans les tableaux 4.1 et 4.2.

Algorithmes	Sans	FT-AIAC-QM	FT-FEC
Temps d'exécution	403s	250s	218s
Gains	–	38%	46%

TABLE 4.1 – Gains en temps de l'exécution d'une application itérative asynchrone sur une architecture de degré d'hétérogénéité 0,45

Algorithmes	Sans	FT-AIAC-QM	FT-FEC
Temps d'exécution	943s	453s	660s
Gains	–	52%	30%

TABLE 4.2 – Gains en temps de l'exécution d'une application itérative asynchrone sur une architecture de degré d'hétérogénéité 0,85

Les résultats décrits dans les deux tableaux précédents montrent clairement que ces algorithmes de placement apportent des gains considérables sur le temps d'exécution de l'application, et ce peu importe le degré d'hétérogénéité de l'architecture d'exécution.

On peut remarquer que pour une architecture peu hétérogène, $hd = 0{,}45$, l'algorithme FT-FEC est plus efficace que FT-AIAC-QM. On remarque qu'au contraire, lorsque l'architecture est hétérogène,

avec $hd = 0{,}85$, c'est l'algorithme FT-AIAC-QM qui obtient les meilleurs gains, alors que l'algorithme FT-FEC chute significativement.

Ces expérimentations indiquent clairement que le choix d'une des deux classes d'algorithmes de placement dépend de l'architecture sur laquelle l'application sera exécutée. On peut en conclure que pour des architectures homogène, avec hd proche de 0, c'est la classe d'algorithmes visant à minimiser l'utilisation de liens externes pénalisants qui est la plus efficace. Lorsque le degré d'hétérogénéité est élevé, avec hd proche de 1, c'est la seconde classe d'algorithmes, ceux visant l'optimisation du temps d'exécution des tâches, qui est plus efficace.

Aux vues de ces résultats, il apparaît qu'il serait intéressant de mettre en œuvre un algorithme de placement qui s'adapte en fonction du degré d'hétérogénéité hd de la plateforme d'exécution. Ainsi, nous proposons dans la section suivante un algorithme hybride s'adaptant au degré d'hétérogénéité de la plateforme d'exécution. L'adaptabilité au degré d'hétérogénéité est d'autant plus important lors de l'utilisation d'une plateforme contenant des machines de calcul volatiles. En effet, de par sa constante évolution, dûe aux arrivées et départs de machines, son degré d'hétérogénéité peut évoluer au cours de l'exécution de l'application. Le fait que l'algorithme de placement soit adaptatif lui permet également de rester efficace lors du choix d'une machine de remplacement.

4.1.4 MAHEVE

Le troisième algorithme de placement que nous présentons est MAHEVE, pour « Mapping Algorithm for HEterogeneous and Volatil Environments », soit « algorithme de placement pour environnements hétérogènes et volatils. ». Cet algorithme est conçu spécialement pour effectuer le placement des tâches d'applications itératives asynchrones exécutées sur des plateformes distribuées, hétérogènes et volatiles.

Comme nous l'avons vu dans les sections 3.3 et 4.1.3, deux classes d'algorithmes de placement existent, chacune permettant d'obtenir de meilleures réductions des temps d'exécution des applications suivant le degré d'hétérogénéité de l'architecture d'exécution. Or, lorsque l'architecture utilisée est volatile, son degré d'hétérogénéité peut évoluer au cours de l'exécution de l'application. Dès lors, le choix de l'algorithme effectué pour le placement initial peut ne plus être le meilleur lorsqu'une panne survient, et qu'il faut choisir une machine remplaçante. C'est pourquoi nous proposons MAHEVE, un algorithme de placement hybride, qui a été conçu pour prendre le meilleur de chacune des deux classes. En effet, les mécanismes de cet algorithme reposent sur le degré d'hétérogénéité de la plateforme d'exécution, et sa politique de placement, mais surtout ses fonctions de remplacement s'adaptent aux variations de celui-ci.

Cet algorithme peut être divisé en deux parties distinctes : le placement initial, et les fonctions pour la tolérance aux pannes.

Placement initial

La première partie de cet algorithme réalise le placement initial des tâches de l'application sur les machines de calcul disponibles dans l'architecture. Comme énoncé précédemment, le degré d'hétérogénéité de la plateforme d'exécution joue ici un rôle important, ainsi que les caractéristiques de l'application. Le placement initial est découpé en trois phases : le classement des clusters, le classement des tâches et enfin l'association des tâches aux machines. Nous présentons tout d'abord l'idée générale de nos classements, puis les différentes étapes menant au placement initial.

Idée générale Comme nous l'avons montré en section 4.1.3, pour être efficace sur tous types d'architecture, la stratégie de placement doit s'adapter au degré d'hétérogénéité de l'architecture d'exécution. Cette adaptation passe par une prise en considération forte du paramètre hd.

Ainsi, notre algorithme base principalement sa stratégie sur ce paramètre pour effectuer les classements décrits dans la suite du document. Comme nous l'avons vu précédemment, deux stratégies s'affrontent et chacune est efficace sur un type d'architecture. Ainsi, notre stratégie repose sur des fonctions partitionnées, aussi bien pour le classement des clusters que pour celui des tâches de calcul. Ainsi, notre algorithme peut reprendre les principes de placements de chacune des deux classes, suivant le cas dans lequel il se trouve (selon degré d'hétérogénéité).

D'après nos expériences, nous avons choisi comme point de partition la valeur $hd = 0,5$. En effet, de par sa définition donnée en section 3.2.1, hd permet de déterminer le degré d'hétérogénéité de la plateforme d'exécution, et sa valeur est comprise en 0 et 1. Il est bien évident que le choix de cette valeur est arbitraire, dans le sens qu'il repose sur l'observation du comportement des algorithmes lors de nos précédentes expériences et que cette valeur peut être affinée. Cependant, une détermination plus fine de ce point de partition n'est pas aisée, car ***le modèle itératif asynchrone étant imprévisible par nature, il est très difficile de mettre en œuvre des simulations, et celles-ci ne seraient pas représentatives***. De même, comme chaque application possède son comportement propre, la réalisation d'expérimentations réelles représenterait un travail considérable, sans compter les aléas dûs aux perturbations subies lors des expérimentations, celles-ci provenant du partage des ressources entre les divers utilisateurs de ces dernières.

Les fonctions de classement reposent sur l'attribution d'une note aux divers éléments constituants le placement ; en l'occurrence les clusters et les tâches. Pour chacun des clusters, une note N_{C_i} lui est attribuée en considérant la puissance de ses machines et le nombre de celles-ci, et pour chaque tâche de l'application une note N_{T_i} lui est attribuée en considérant d'une part sa charge de calcul et d'autre part son nombre de dépendances (ou voisins de calcul).

Le principe de notation des clusters repose sur l'idée qu'un cluster obtient une meilleure note lorsque :

- $hd < 0{,}5$ et qu'il contient plus de machines disponibles que les autres, soit nd_i (nombre de machines disponibles) ;
- $hd \geq 0{,}5$ et qu'il contient des machines disponibles plus puissantes que les autres, soit $\overline{P_{Cdi}}$ (puissance moyenne des machines disponibles).

La figure 4.1 montre un graphique représentant des notes obtenues par deux clusters, suivant la variation de hd, c'est-à-dire suivant l'architecture dans laquelle ils se trouvent.

Les caractéristiques des deux clusters sont :

- ClusterPuissant (noté cp) : $P_{Cd_{cp}} = 100$ (puissance totale du cluster), $nd_{cp} = 15$ (nombre de machines disponibles), $\overline{P_{Cd_{cp}}} = 6{,}67$ (puissance moyenne des machines disponibles) ;
- ClusterNombre (noté cn) : $P_{Cd_{cn}} = 100$, $nd_{cn} = 17$, $\overline{P_{Cd_{cn}}} = 5{,}88$.

En considérant les caractéristiques de ces deux clusters, on remarque qu'ils possèdent la même puissance totale, soit $P_{Cd_i} = 100$, mais possèdent un nombre de machines différentes, soit $nd_{cp} = 15$ pour le ClusterPuissant (noté cp) et $nd_{cn} = 17$ pour le ClusterNombre (noté cn). Ces informations nous indiquent que les clusters possèdent des machines de puissances différentes. On peut alors remarquer que lorsque hd est proche de 0, le cluster possédant le plus grand nombre de machines disponibles est favorisé, alors que lorsque hd est proche de 1 c'est le cluster possédant les machines disponibles les plus puissantes qui est favorisé. Il est à noter ici que lorsque deux clusters ont une même puissance, au niveau du point de bascule $hd = 0{,}5$, l'algorithme donne la préférence au cluster ayant les machines les plus puissantes.

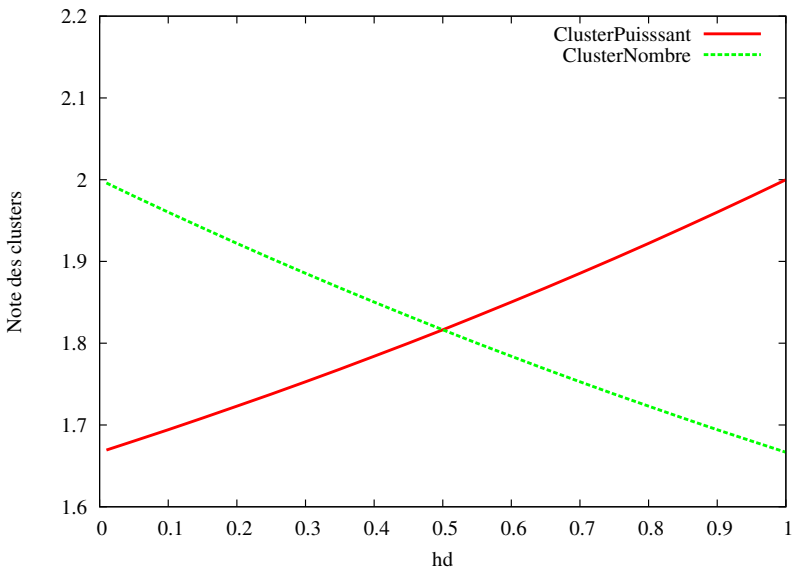

FIGURE 4.1 – Exemple de l'évolution des notes de 2 clusters dans une architecture

Ensuite, concernant la notation des tâches de calcul, le principe ici est également de favoriser certaines caractéristiques. Ainsi une tâche de calcul obtient une meilleure note lorsque :

- $hd < 0{,}5$ et que la tâche possède beaucoup de dépendances. Ainsi la localité des tâches est favorisée, car les clusters favorisés dans ce cas sont ceux possédant le plus grand nombre de machines disponibles ;
- $hd \geq 0{,}5$ et que la tâche possède une charge de calcul élevée. Ainsi, comme dans ce cas les clusters favorisés sont ceux possédant les machines disponibles les plus puissantes, ces tâches pourront s'exécuter plus rapidement.

La fonction de notation des tâches doit suivre la même forme et obtenir le même comportement que celle pour la notation des clusters. Le but étant clairement d'obtenir des classements cohérents et de garder une ligne directrice commune entre les deux classements.

De ce fait, ces deux fonctions permettent à l'algorithme de définir une stratégie de placement en accord avec le degré d'hétérogénéité de la plateforme d'exécution.

Classement des clusters Dans un premier temps, afin de réaliser le placement initial des tâches de calcul sur les machines, l'algorithme effectue un classement des clusters. Il est à noter ici que nous considérons un cluster comme un ensemble de machines similaires, c'est-à-dire ayant les mêmes caractéristiques, reliées par un réseau à très faible latence. Le classement des clusters s'opère en deux étapes. La première consiste à donner une note à chacun des clusters, en fonction de leur nombre de machines disponibles ainsi que de leur puissance. Ici ne sont considérées que les machines disponibles, c'est-à-dire celles qui ne sont pas déjà impliquées dans un calcul. Ensuite, dans l'optique de toujours privilégier la localité des tâches de l'application, permettant de réduire les temps de communication et d'augmenter la vitesse de convergence, les clusters sont triés suivant une métrique axée sur le degré d'hétérogénéité de la plateforme. L'algorithme 5 présente ce classement.

Dans un premier temps, chaque cluster C_i présent dans la liste lc reçoit une note N_{C_i}, en relation avec son nombre de machines et leur puissance, en fonction du degré d'hétérogénéité hd (lignes 1 à 3). La note N_{C_i} obtenue par un cluster dépend ainsi du degré d'hétérogénéité de la plateforme hd, du nombre de machines disponibles dans le cluster nd_i, et de leur puissance moyenne $\overline{P_{Cd_i}}$. Comme nd_i et

Algorithme 5 : Algorithme de MAHEVE : classement des clusters

Entrées : *lc* (liste des clusters), *hd* (degré d'hétérogénéité de la plateforme)
Sorties : *clustersTriés* (liste des clusters triée)

1 **pour** $i = 1$ *à* $|C|$ **faire**
2 | $notesCluster[i] \leftarrow N_{C_i}$
3 **fin**
4 **pour** $i = 1$ *à* $|C| - 1$ **faire**
5 | **pour** $k = i + 1$ *à* $|C|$ **faire**
6 | | insérer dans *associationsClusters* le couple $\{C_i, C_k\}$
7 | **fin**
8 **fin**
9 **pour** $i = 1$ *à* $taille(associationsClusters)$ **faire**
10 | **si** $meilleurCluster(associationsClusters(i))$ *non présent dans clustersTriés* **alors**
11 | | ajouter à *clustersTriés* $meilleurCluster(associationsClusters(i))$
12 | **fin**
13 | **si** $pireCluster(associationsClusters(i))$ *non présent dans clustersTriés* **alors**
14 | | ajouter à *clustersTriés* $pireCluster(associationsClusters(i))$
15 | **fin**
16 **fin**
17 renvoyer *clustersTriés*

$\overline{P_{Cd_i}}$ ne sont pas du même ordre de grandeur, nous les normalisons en les divisant par leur plus grande valeur respective. Nous notons Nnd_i la normalisation de nd_i avec $Nnd_i = \frac{nd_i}{max(nd_i)}$, et $N\overline{P_{Cd_i}}$ celle de $\overline{P_{Cd_i}}$ avec $N\overline{P_{Cd_i}} = \frac{N\overline{P_{Cd_i}}}{max(N\overline{P_{Cd_i}})}$. La note d'un cluster C_i est la suivante :

$$N_{C_i} = N\overline{P_{Cd_i}}^{hd} + {Nnd_i}^{1-hd}. \tag{4.1}$$

Cette formule nous permet donc de satisfaire les critères énoncés précédemment, suivant le degré d'hétérogénéité de la plateforme. Afin de montrer que nos critères énoncés précédemment sont satisfaits, il suffit d'étudier les limites de cette équation qui sont :

- $\lim_{hd \to 0} N_{C_i} = Nnd_i + 1$;
- $\lim_{hd \to 1} N_{C_i} = N\overline{P_{Cd_i}} + 1$.

Ainsi, pour une plateforme homogène (avec $hd < 0{,}5$) les clusters avec le plus grand nombre de machines disponibles seront privilégiés ($\lim_{hd \to 0} N_{C_i} = Nnd_i + 1$), et inversement, pour une plateforme hétérogène (avec $hd \geq 0{,}5$) ce sont les clusters possédant les machines les plus puissantes qui seront privilégiés ($\lim_{hd \to 1} N_{C_i} = N\overline{P_{Cd_i}} + 1$).

Une fois les clusters notés, l'idée est d'essayer de trouver les clusters les mieux notés ayant des liens de communication fournissant une latence faible (nous rappelons ici que nous utilisons dans notre étude la latence des liens, et non leur débit). Cette mise en couple est effectuée entre les lignes 4 et 8. Chaque couple, noté $\{C_i, C_j\}$, possède une note $NCouple_{ij}$. Cette note est calculée en utilisant la formule :

$$NCouple_{ij} = \frac{N_{C_i} + N_{C_j}}{ll_{ij}} \tag{4.2}$$

dans laquelle N_{C_i} et N_{C_j} représentent les notes respectives des deux clusters, et ll_{ij} la latence existante entre eux. Nous utilisons ici la notation ll_{ij} représentant la latence existante sur un lien de

communication entre deux machines M_i et M_j, comme la latence existante entre les deux clusters C_i et C_j auxquels appartiennent les machines M_i et M_j. En effet nous considérons que toutes les machines d'un même cluster utilisent le même lien de communication pour échanger des données avec des machines d'un autre cluster. Chaque couple est inséré dans la liste $associationsClusters$. Cette liste étant triée dans l'ordre des associations ayant la meilleure note à celles ayant la moins bonne, l'insertion d'une nouvelle association s'effectue directement à sa place.

Une fois toutes les associations constituées et triées, le classement final des clusters peut être effectué (lignes 9 à 16). Les associations sont prises dans leur ordre de tri et le cluster ayant la meilleure note N_{C_i}, sélectionné dans l'association à l'aide la fonction *meilleurCluster()*, est ajouté en premier dans le classement final *clustersTriés* si celui-ci n'y est pas présent. Ensuite, l'autre cluster composant l'association (celui ayant obtenu la moins bonne note N_{C_i}), choisi par la fonction *pireCluster()*, est ajouté dans le classement final si celui-ci n'y est pas présent. Enfin, la liste triée des clusters *clustersTriés* est renvoyée (ligne 17).

Ce classement des clusters permet d'obtenir une liste triée de ceux-ci suivant leurs caractéristiques, pondérées par le degré d'hétérogénéité de l'architecture, et leur proximité. La décomposition de la complexité de ce classement est de $O(|C|)$ pour l'attribution des notes, de $O(|C| \times |C-1| \times log|C|)$ pour la mise en couple, et de $O(\frac{|C| \times |C-1|}{2})$ pour le tri final. Ainsi, sa complexité est de $O(|C|^2 \times log|C|)$.

Classement des tâches de calcul La seconde phase du placement initial, après le classement des clusters, est le classement des tâches de l'application. Les tâches sont triées suivant les mêmes principes que pour les clusters. L'algorithme 6 présente ce classement.

Algorithme 6 : Algorithme de MAHEVE : classement des tâches de calcul

Entrées : *lt* (liste des tâches), *hd* (degré d'hétérogénéité de la plateforme)
Sorties : *tâchesTriées* (liste des tâches triée)

```
1  tmp ← nouvelle liste de tâches (vide)
2  pour i = 1 à |T| faire
3      calculer N_{T_i}
4      insérer T_i dans premierTri
5  fin
6  tant que taille(premierTri) > 0 faire
7      si tmp est vide alors
8          ajouter dans tmp la tâche tête(premierTri)
9          supprimer tête(premierTri)
10     fin
11     tant que tmp est non vide faire
12         déplacer tête(tmp) dans tâchesTriées
13         nbDepVoulues ← nombre de dépendances voulues (formule (4.4))
14         pour i = 1 à nbDepVoulues faire
15             ajouter dans tmp la prochaine dépendance de la tâche traitée présente dans
               premierTri
16             enlever cette tâche de premierTri
17         fin
18     fin
19 fin
20 renvoyer tâchesTriées
```

Tout comme pour les clusters, les tâches reçoivent une note N_{T_i} dans laquelle le degré d'hétérogénéité hd intervient (ligne 3). Cette note N_{T_i} est obtenue suivant la formule :

$$N_{T_i} = ct_i^{hd} \times dt_i^{1-hd} \tag{4.3}$$

dans laquelle ct_i représente la charge de calcul de la tâche T_i et dt_i son nombre de dépendances. Cette formule correspond à nos attentes car une étude aux limites de l'équation donne :

- $\lim_{hd \to 0} N_{T_i} = dt_i$;
- $\lim_{hd \to 1} N_{T_i} = ct_i$.

Il est à noter ici que contrairement à la notation des clusters, nous utilisons ici une multiplication car le second membre dt_i est une quantité sans unité (le nombre de dépendances), ceci n'impliquant pas une normalisation des variables. Cependant, si nous avions normalisé ces deux variables, la formule obtenue aurait été identique à celle utilisée pour la notation des clusters.

Une fois la note obtenue, chaque tâche est insérée dans une liste triée $premierTri$, directement à sa place (ligne 4). Ensuite, le classement définitif des tâches est effectué (lignes 6 à 19). Ce classement sera disponible dans la liste *tâchesTriées*. L'algorithme entre alors dans une boucle (lignes 6 à 19) dont il ne sortira qu'une fois que toutes les tâches auront été traitées. Une liste temporaire *tmp* est utilisée. Cette dernière sert de tampon de tri pour le classement final. Tout d'abord, si cette liste est vide, la première tâche présente dans $premierTri$ (celle ayant la meilleure note) est retirée de cette liste et est ajoutée dans *tmp* (lignes 7 à 10). Ensuite l'algorithme rentre dans une boucle sur la liste *tmp* (lignes 11 à 18). La première tâche dans la liste *tmp* est choisie puis insérée dans la liste finale *tâchesTriées* (ligne 12). Ensuite $nbDepVoulues$ est calculé (ligne 13). Ce nombre représente la quantité de dépendances de la tâche en cours de traitement qui seraient « mieux placées » en étant affectées à des machines du même cluster qu'elle. Ce calcul dépend lui aussi de hd :

$$nbDepVoulues = dt_i \times (1 - hd) + 1. \tag{4.4}$$

Ainsi, plus hd est petit, plus le nombre de dépendances qui seraient « mieux placées » au plus proche de la tâche courante est élevé, et inversement lorsque hd est grand, ce nombre de dépendances est faible. Une fois ce nombre de dépendances déterminé, l'algorithme cherche les dépendances dans la liste $premierTri$, en la parcourant dans l'ordre. Chaque tâche dépendante est alors retirée de $premierTri$ (ligne 16) et ajoutée dans la liste *tmp* pour qu'elles soient traitées (ligne 15). Ce mécanisme permet de préserver la localité des tâches interdépendantes, en accord avec le classement des clusters.

Enfin, une fois toutes les tâches traitées, la liste triée des tâches *tâchesTriées* est renvoyée (ligne 20). La décomposition de la complexité de ce classement des tâches est de $O(|T| \times log|T|)$ pour la notation et le premier classement de celles-ci, de $O(|T| \times |T| \times log|T|)$ pour leur classement final. Ainsi, la complexité de ce classement est de $O(|T|^2 \times log|T|)$.

Association des tâches aux machines Enfin, la dernière phase du placement initial est l'association des tâches aux machines de calcul. Cette partie est fortement inspirée de la méthode d'association utilisée par les algorithmes visant à minimiser l'utilisation des liens externes. En effet, le but ici est de remplir au maximum les clusters. Cependant, nous introduisons ici une variation de ce processus, nous permettant d'intégrer dès le placement initial des facilités pour la tolérance aux pannes, en réservant quelques machines – ainsi le cluster n'est pas totalement rempli. Le fait de « réserver » une machine n'indique pas qu'elle est « bloquée » pour l'exécution d'une application, mais simplement qu'elle n'est pas utilisée dans le placement initial. L'algorithme 7 présente ce mécanisme d'association.

Algorithme 7 : Algorithme de MAHEVE : association des tâches aux machines

Entrées : *clustersTriés* (liste triée des clusters), *tâchesTriées* (liste triée des tâches), *nbSauv* (nombre de machines réservées sur chaque cluster), *hd* (degré d'hétérogénéité)

Sorties : *mp* (placement des tâches)

```
1  ind ← 1 ; div ← 1
2  pour i = 1 à |T| faire
3      ok ← faux
4      tant que ok = faux faire
5          calcul de nbDep avec l'équation (4.5) (utilisant la variable div)
6          changeCluster ← faux
7          si machinesLibres(clustersTriés[ind]) − nbSauv > nbDep alors
8              ajouter à mp l'association (machineLibre(clustersTriés[ind]),tâchesTriées[i])
9              ok ← vrai
10         sinon
11             changeCluster ← vrai
12         fin
13         si changeCluster = vrai alors
14             ind ← ind + 1
15             si ind = |C| alors
16                 ind ← 1
17                 div ← div + 1
18             fin
19         fin
20     fin
21 fin
22 renvoyer mp
```

Dans cette partie de l'algorithme, nous utilisons les listes triées des clusters et des tâches qui ont été réalisées dans les deux phases précédentes. Un paramètre supplémentaire *nbSauv*, renseigné par l'utilisateur, indique combien de machines doivent être réservées sur chaque cluster dans cette phase d'association. Ces machines seront utilisées lors du remplacement d'une machine tombée en panne. Ce mécanisme offre la possibilité de préserver dans une certaine mesure la localité des tâches, même en cas de pannes.

L'algorithme entre dans une boucle sur les tâches (lignes 2 à 20). Pour chaque tâche T_i est calculé un nombre de dépendances *nbDep* qui seraient « mieux placées » sur des machines du même cluster que la tâche courante (ligne 5). Ce nombre est déterminé par la formule :

$$nbDep = \frac{dt_i \times (1 - hd)}{div} + 1 \tag{4.5}$$

dans laquelle dt_i représente le nombre de dépendances de T_i, et *div* est une variable de contrôle de ce nombre de dépendances. Cette variable, initialisée à 1 (ligne 1), est utilisée lorsque le mécanisme d'association n'a pas pu aboutir. En effet, si toutes les tâches n'ont pas pu être placées et que tous les clusters ont été parcourus, cette variable permet d'abaisser le nombre de dépendances requises, diminuant ainsi le nombre de machines disponibles requis pour le placement des tâches. Ainsi, si le cluster courant possède un nombre suffisant de machines disponibles, c'est-à-dire supérieur à *nbDep*, tout en gardant *nbSauv* machines en réserve (ligne 7), alors la tâche T_i est associée à une machine

de ce cluster. Cette association est ajoutée dans le placement final (ligne 8). Dans le cas contraire, l'algorithme choisit de changer de cluster en prenant le suivant dans la liste (lignes 10 à 14). Si tous les clusters ont été utilisés, la liste est reprise depuis le début (ligne 16), et la variable de contrôle *div* est incrémentée (ligne 17). Une fois que toutes les tâches ont été affectées à une machine d'un cluster, le placement final *mp* est renvoyé (ligne 22).

La complexité de cette affectation, pour l'obtention du placement initial se calcule suivant : si l'on considère une application avec un schéma de communication complet et que chaque cluster possède une seule machine, alors on obtient une complexité de $O(|T| \times |C| \times |T|)$, soit $O(|T|^2)$. Ainsi, la complexité totale de l'algorithme de placement est de $O(|C|^2 + |T|^2)$.

Fonctions pour la tolérance aux pannes

La seconde partie de l'algorithme concerne les fonctions pour la tolérance aux pannes. Les deux fonctions présentées ici permettent pour la première de choisir une machine de remplacement lors de la détection d'une panne, et la seconde est utilisée pour choisir des machines sur lesquelles seront déployées des répliques des entités de gestion de l'environnement d'exécution.

Remplacement de machines Comme énoncé dans les sections précédentes, un algorithme de placement utilisé pour l'exécution d'applications sur des plateformes volatiles doit fournir une politique de remplacement efficace. Le principe qui est utilisé dans cet algorithme est simple. Lors de la création du placement initial, décrit dans la section précédente, un certain nombre de machines ont été réservées, spécialement pour la tolérance aux pannes, dans chaque cluster.

Lorsqu'une machine tombe en panne, l'algorithme cherche une machine disponible dans le même cluster. Cette machine peut faire partie de celles qui ont été réservées, ou bien être de celles qui ont rejoint la plateforme au cours de l'exécution de l'application. Si une telle machine est disponible, alors elle sera utilisée pour effectuer le remplacement de celle venant de tomber en panne. S'il n'y a plus de machine de disponible, ceci signifie que la plateforme a subi de nombreux changements. Dans ce cas, afin d'ajuster les choix des machines, la liste triée des clusters *clustersTriés* est mise à jour. Ce mécanisme permet à l'algorithme de garder une vue de la plateforme la plus exacte possible. Une fois la mise à jour effectuée, l'algorithme renvoie une machine disponible dans le cluster le mieux noté. Dans le cas où des machines se connectent à la plateforme au cours de l'exécution de l'application, celles-ci viennent compléter le cluster correspondant, ajoutant ainsi de nouvelles machines disponibles – au même titre que celles réservées.

Choix de machines pour la gestion de l'environnement d'exécution Afin de pouvoir exécuter des applications sur des architectures volatiles, les environnements d'exécution, comme par exemple JaceP2P-V2, répliquent leurs entités de gestion. Ceci leur permet non seulement de pouvoir répartir les charges de travail afin de passer à l'échelle, mais surtout de résister aux pannes de telles entités.

Comme ces entités sont généralement répliquées sur des machines de calcul composant l'architecture, nous proposons une fonction permettant de choisir ces machines. Le but ici est de ne pas choisir au hasard les machines utilisées pour la gestion de l'environnement, en risquant de pénaliser l'application. Les machines choisies pour l'environnement d'exécution sont prises sur les clusters où le nombre de machines disponibles est supérieur au paramètre *nbSauv*. De ce fait, ceci permet de ne pas utiliser les machines destinées à la tolérance aux pannes, tout en assurant une certaine proximité entre les machines de calcul et celles de gestion.

4.2 Intégration dans JaceP2P-V2

Dans cette section nous présentons l'intégration de ces algorithmes de placement au sein de l'environnement JaceP2P-V2. Le but recherché ici n'est pas de simplement ajouter ces trois algorithmes au sein de la plateforme, mais de lui permettre de les utiliser, et à l'avenir de pouvoir facilement en intégrer d'autres. C'est pourquoi une bibliothèque a été créée. Cette bibliothèque est donc un élément à part de la plateforme d'exécution, permettant le développement et l'utilisation d'algorithmes de placement.

Dans un premier temps nous présentons l'architecture, le fonctionnement et les possibilités de cette bibliothèque. Puis nous présentons son intégration dans l'environnement JaceP2P-V2.

4.2.1 Bibliothèque « Mapping »

Cette bibliothèque, écrite en Java, est libre et à sources ouvertes [1], ce qui lui permet d'être utilisée par tout un chacun, afin de l'améliorer et de la compléter.

Cette bibliothèque permet de modéliser aussi bien les applications que les architectures d'exécution. Dans sa version actuelle, elle ne permet de modéliser les applications que sous forme de TIG.

Pour la modélisation de l'architecture d'exécution, plusieurs possibilités sont offertes. En effet, il est possible d'utiliser le modèle en trois couches (architecture, clusters et machines). Chaque entité est représentée par une classe propre. Dans le cas de notre étude nous utilisons cette vue de l'architecture. La classe des machines possède les attributs suivants : son nom, le nombre de processeurs, le nombre de cœurs de calcul, la fréquence des cœurs, et la quantité de mémoire. Il est possible de renseigner un attribut supplémentaire, qui peut être utilisé pour assurer un lien avec l'objet représentant la machine dans l'environnement d'exécution. Concernant les clusters, ceux-ci tiennent à jour une liste des machines qui leur sont associées. Au niveau de l'architecture sont rassemblés les clusters ainsi que la liste globale de toutes les machines. C'est aussi dans ce niveau que se retrouvent les caractéristiques des liens existants entre les clusters.

Toutes ces informations permettent de réaliser des algorithmes de placement. Il suffit de créer une nouvelle classe dérivant du type abstrait « Algo ». Dès lors, l'utilisateur peut utiliser toutes les opérations disponibles, comme connaître le nombre de machines disponibles dans les clusters ou la puissance de celles-ci, afin de construire un placement.

La bibliothèque fournit aussi quelques méthodes supplémentaires. Parmi ces méthodes, on peut citer l'importation et l'exportation des graphes et des placements au format XML. Ceci permet d'utiliser les différents graphes, voire même le résultat d'un algorithme de placement, afin d'effectuer des traitements sur ceux-ci à l'aide d'outils externes. Ainsi l'utilisation de cette bibliothèque n'est pas uniquement réservée à des environnements d'exécution. D'autres fonctions permettent également de renseigner automatiquement les informations de l'architecture d'exécution. En effet, il est possible pour un démon de créer une représentation de sa machine lors de son lancement, en utilisant par exemple l'application Linpack [53] afin de déterminer la puissance de calcul de la machine, ou utiliser les mécanismes Java pour déterminer le nombre de processeurs disponibles ainsi que leur fréquence.

Cette bibliothèque fournit donc tous les outils nécessaires à l'élaboration de placements pour l'exécution d'applications itératives asynchrones sur des architectures hétérogènes et volatiles. Bien que fournissant tout ce dont nous avons besoin pour notre étude, elle peut être complétée afin de satisfaire d'autres usages.

1. Le dépôt de développement de la bibliothèque Mapping est disponible à l'adresse `http://info.iut-bm.univ-fcomte.fr/pub/gitweb/`.

4.2.2 Modifications de JaceP2P-V2

La bibliothèque « Mapping » présentée dans la section précédente est intégrée directement dans la plateforme JaceP2P-V2. Les diverses fonctions de cette bibliothèque sont réparties sur plusieurs entités de l'environnement. La figure 4.2 illustre cette répartition.

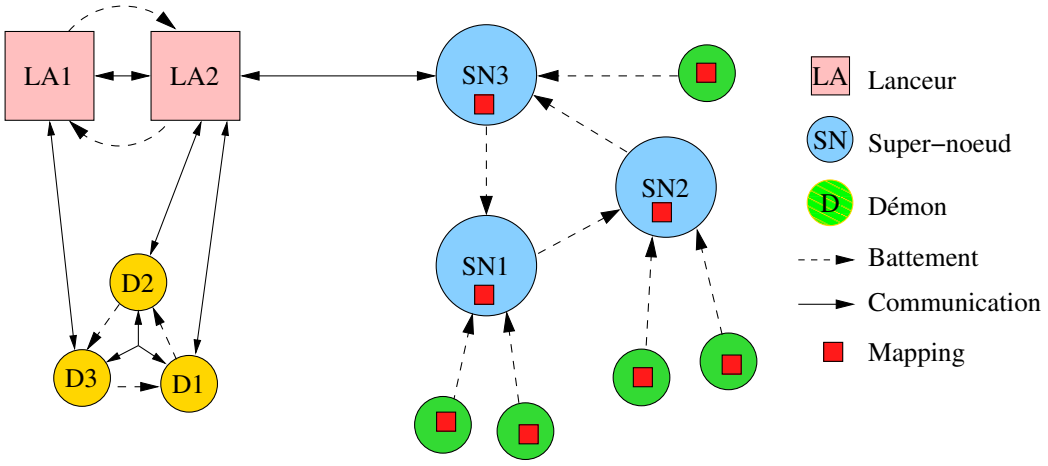

FIGURE 4.2 – Intégration de la bibliothèque Mapping dans JaceP2P-V2

Comme on peut le remarquer, les éléments concernés sont les super-nœuds et les démons. Les modifications sur chacun de ces éléments sont les suivantes :

- sur les démons : lorsqu'un démon se connecte à un super-nœud, il exécute une méthode de la bibliothèque lui permettant de créer une représentation de la machine sur laquelle il est exécuté. Cette représentation contient le nombre et la fréquence des cœurs de calcul disponibles, ainsi que la quantité de mémoire disponible ;
- sur les super-nœuds : ils sont en charge d'avoir une vue globale de l'architecture. En effet, c'est auprès d'eux que se connectent les machines disponibles. Chaque super-nœud contient une liste de tous les démons connectés à la plateforme. Cette liste est utilisée pour générer le graphe d'architecture.

Lors de son exécution, le lanceur effectue une demande de démons auprès d'un super-nœud, en lui spécifiant le nombre de démons qui seront dédiés au calcul, ainsi que l'algorithme de placement qui doit être utilisé et les caractéristiques de l'application. Le super-nœud crée alors un graphe de l'architecture courante et un graphe représentant l'application. Ensuite il exécute l'algorithme de placement choisi et renvoie le placement effectué au lanceur, avec un identifiant. Cet identifiant permet au super-nœud de pouvoir faire référence au placement et à l'algorithme choisi dans le cas où une panne surviendrait durant l'exécution de l'application.

Étant donné que l'environnement JaceP2P-V2 est totalement tolérant aux pannes, en utilisant des réplications de ses entités de gestion, quelques modifications ont dû être effectuées afin de préserver cette caractéristique. En effet, tous les super-nœuds sont informés de chaque modification de la plateforme (connexions et déconnexions de machines). Prenons l'exemple de trois super-nœuds. Sur le premier une machine se connecte, et sur le second une autre se déconnecte. Des mécanismes ont été mis en place afin que les trois super-nœuds puissent maintenir une vue à jour de la plateforme, tout en gardant la répartition de la charge de surveillance. Le fait d'être informé d'un changement de la plateforme permet de mettre à jour les graphes d'architecture présents dans les placements dont les applications sont en cours d'exécution. Ainsi, lorsqu'une panne survient, la fonction de remplacement de l'algorithme travaille sur une vue en temps réel de l'architecture. Toujours pour la tolérance aux pannes, lorsqu'un

super-nœud effectue un placement, il le renvoie non seulement au lanceur ayant effectué la demande, mais aussi aux autres super-nœuds, afin que ceux-ci puissent prendre la relève s'il tombe en panne. Concernant les lanceurs, ils disposent maintenant d'un appel supplémentaire sur le super-nœud, leur permettant de réserver des machines supplémentaires qui leur serviront pour leur réplication.

Ainsi, la bibliothèque « Mapping » est totalement intégrée à l'environnement JaceP2P-V2. Elle lui permet d'utiliser divers algorithmes de placement tout en préservant les qualités essentielles de l'environnement, qui sont la totale tolérance aux pannes et le passage à l'échelle.

4.3 Conclusion

Dans ce chapitre nous avons présenté nos contributions au placement des tâches d'applications itératives asynchrones sur des architectures d'exécution hétérogènes et volatiles. Nous avons exposé trois algorithmes de placement et leur intégration dans l'environnement JaceP2P-V2.

Concernant les algorithmes de placement, nous avons détaillé trois solutions différentes. Les deux premiers algorithmes, FT-FEC et FT-AIAC-QM, sont deux adaptations d'algorithmes existants. Le premier fait partie de la classe des algorithmes visant à réduire l'utilisation des liens pénalisant d'une architecture en effectuant une clusterisation des tâches. Nous avons ajouté à cet algorithme une fonction permettant la sélection de machines de remplacement lorsque des machines impliquées dans le calcul tombent en panne. Le second algorithme fait partie de la classe des algorithmes cherchant à réduire le temps d'exécution des tâches de l'application, en utilisant les machines les plus puissantes. Nous avons modifié cet algorithme, dont une partie utilisait un placement de plusieurs tâches sur une même machine de calcul, dans le but de réduire le coût des communications. Comme ce placement multiple n'est pas autorisé dans notre modèle, nous l'avons remplacé par une recherche de proximité des machines sur lesquelles sont placées deux tâches interdépendantes. Nous lui avons aussi ajouté une fonction permettant de choisir des machines de remplacement lors de pannes de machines de calcul. Pour rappel, nous avons voulu garder l'esprit de ces deux algorithmes lors de nos modifications afin de les rendre compatibles avec nos contraintes, imposées par notre modèle.

Le troisième algorithme que nous avons présenté, MAHEVE, représente notre principale contribution. Les deux précédentes classes d'algorithmes de placement ne sont efficaces que sur un type d'architecture. Ce nouvel algorithme a été conçu en prenant le meilleur des deux classes, lui permettant d'être exécuté sur n'importe quelle architecture. L'algorithme est basé sur le degré d'hétérogénéité de l'architecture d'exécution, lui conférant ainsi un caractère adaptatif. Il prend également en compte les caractéristiques de l'application, principalement les dépendances existantes entre les tâches de calcul, afin d'ajuster au mieux le placement initial. Un point important est qu'il permet également de laisser une marge de sécurité en prévision de pannes durant l'exécution de l'application. Cette marge peut être réglée par l'utilisateur. Deux fonctions dédiées à la tolérance aux pannes sont également incluses. La première est utilisée lorsqu'une machine tombe en panne en cours de calcul, afin de lui retrouver une remplaçante. La seconde permet de sélectionner des machines servant à l'environnement d'exécution qui les utilise afin d'être tolérant aux pannes et de passer à l'échelle.

Nous avons également présenté l'intégration de ces algorithmes de placement au sein de l'environnement JaceP2P-V2. Tout d'abord nous avons présenté notre bibliothèque « Mapping », qui permet l'élaboration d'algorithmes de placement. Cette bibliothèque peut être améliorée et complétée afin d'ouvrir son champ d'application. Pour finir, nous avons présenté l'intégration de cette bibliothèque au sein de l'environnement JaceP2P-V2. Cette intégration lui permet donc d'utiliser divers algorithmes de placement. Nous avons tout particulièrement pris soin de préserver les points forts de l'environnement, qui sont la totale tolérance aux pannes et le passage à l'échelle, en mettant en place des mécanismes de réplication des placements effectués.

Chapitre 5

Expérimentations

Dans ce chapitre nous présentons les expérimentations que nous avons menées afin de vérifier l'efficacité de nos algorithmes de placement présentés dans le chapitre précédent. La nature du modèle itératif asynchrone, le rendant imprévisible notamment concernant le nombre d'itérations qui seront effectuées, rend difficile l'utilisation de simulations. C'est pourquoi nous avons mené des expérimentations en exécutant des applications réelles sur des architectures réelles.

Dans un premier temps nous présentons les conditions de ces expérimentations. Nous commençons tout d'abord par la description des architectures d'exécution qui ont été utilisées, puis sont présentées les deux applications qui ont été exécutées. Dans un second temps nous présentons les résultats de nos expérimentations. Dans l'expérimentation exécutant la seconde application, nous présentons la modification d'un paramètre de l'algorithme de placement MAHEVE que nous avons dûe effectuer à la suite des premières expériences que nous avons menées avec cette application.

5.1 Conditions d'expérimentation

Cette section a pour objectif de présenter les conditions de nos expérimentations. Dans un premier temps nous donnons les caractéristiques des architectures d'exécution que nous avons utilisées. Ensuite nous décrivons le paramétrage des algorithmes de placement. Enfin, nous présentons les deux applications qui ont été utilisées : l'application de la multidécomposition avec gradient conjugué, et celle de la résolution de l'équation d'advection-diffusion en trois dimensions.

5.1.1 Architectures d'exécution

Les architectures utilisées sont des ensembles de clusters et de machines faisant partie de la grille d'expérimentation Grid'5000. Sa description est donnée en section 2.1.3. Il est à noter que les valeurs du degré d'hétérogénéité de chaque plateforme tiennent compte de la puissance des machines qui est déterminée en utilisant l'application Linpack. Ceci permet de déterminer avec plus de précision la puissance réelle et disponible de chaque machine. Ainsi ne sont données ici que les caractéristiques des machines, ne reflétant pas forcément la réalité de la puissance disponible pour chacune d'entre elles.

Pour la réalisation de nos expériences, nous avons utilisé au total 6 architectures, soit 3 pour chaque application. Chaque architecture a été sélectionnée avec des degrés d'hétérogénéité différents,

permettant d'observer l'efficacité des algorithmes de placement, en fonction de ceux-ci. Pour la première application, les architectures sont les suivantes :

- architecture 1.1 : elle est composée d'un total de 115 machines réparties sur 4 clusters de 4 sites. Les détails de sa composition sont : 30 machines du cluster *bordereau* à Bordeaux (4 cœurs de calcul à 2,6GHz, 4Go de RAM), 30 machines du cluster *sol* à Sophia-Antipolis (4 cœurs de calcul à 2,6GHz, 4Go de RAM), 25 machines du cluster *chicon* à Lille (4 cœurs de calcul à 2,6GHz, 4Go de RAM), et 31 machines du cluster *pastel* à Toulouse (4 cœurs de calcul à 2,6GHz, 4Go de RAM). Une machine du cluster *pastel* a été réservée pour l'exécution du super-nœud. Cette architecture présente un degré d'hétérogénéité *hd* de 0,08. Cette architecture est donc considérée comme homogène ;
- architecture 1.2 : elle est composée d'un total de 110 machines réparties sur 4 clusters de 3 sites. Les détails de sa composition sont : 35 machines du cluster *bordereau* à Bordeaux (4 cœurs de calcul à 2,6GHz, 4Go de RAM), 30 machines du cluster *paramount* à Rennes (4 cœurs de calcul à 2,33GHz, 8Go de RAM), 20 machines du cluster *paraquad* à Rennes (4 cœurs de calcul à 2,33GHz, 4Go de RAM), et 26 machines du cluster *pastel* à Toulouse (4 cœurs de calcul à 2,6GHz, 4Go de RAM). Une machine du cluster *pastel* a été réservée pour l'exécution du super-nœud. Cette architecture présente un degré d'hétérogénéité *hd* de 0,5. Cette architecture est donc considérée comme neutre ;
- architecture 1.3 : elle est composée d'un total de 115 machines réparties sur 5 clusters de 4 sites. Les détails de sa composition sont : 30 machines du cluster *bordereau* à Bordeaux (4 cœurs de calcul à 2,6GHz, 4Go de RAM), 25 machines du cluster *paramount* à Rennes (4 cœurs de calcul à 2,33GHz, 8Go de RAM), 25 machines du cluster *paradent* à Rennes (8 cœurs de calcul à 2,5GHz, 32Go de RAM), 15 machines du cluster *genepi* à Grenoble (8 cœurs de calcul à 2,5GHz, 8Go de RAM), et 21 machines du cluster *pastel* à Toulouse (4 cœurs de calcul à 2,6GHz, 4Go de RAM). Une machine du cluster *pastel* a été réservée pour l'exécution du super-nœud. Cette architecture présente un degré d'hétérogénéité *hd* de 0,72. Cette architecture est donc considérée comme hétérogène.

Pour la seconde application, les architectures utilisées sont les suivantes :

- architecture 2.1 : elle est composée d'un total de 113 machines réparties sur 4 clusters de 3 sites. Les détails de sa composition sont : 30 machines du cluster *bordereau* à Bordeaux (4 cœurs de calcul à 2,6GHz, 4Go de RAM), 30 machines du cluster *sol* à Sophia-Antipolis (4 cœurs de calcul à 2,6GHz, 4Go de RAM), 23 machines du cluster *chicon* à Lille (4 cœurs de calcul à 2,6GHz, 4Go de RAM), et 31 machines du cluster *helios* à Sophia-Antipolis (4 cœurs de calcul à 2,2GHz, 4Go de RAM). Une machine du cluster *helios* a été réservée pour l'exécution du super-nœud. Cette architecture présente un degré d'hétérogénéité *hd* de 0,05. Cette architecture est donc considérée comme homogène ;
- architecture 2.2 : elle est composée d'un total de 109 machines réparties sur 4 clusters de 3 sites. Les détails de sa composition sont : 20 machines du cluster *bordereau* à Bordeaux (4 cœurs de calcul à 2,6GHz, 4Go de RAM), 35 machines du cluster *sol* à Sophia-Antipolis (4 cœurs de calcul à 2,6GHz, 4Go de RAM), 25 machines du cluster *graphene* à Nancy (4 cœurs de calcul à 2,53GHz, 16Go de RAM), et 30 machines du cluster *helios* à Toulouse (4 cœurs de calcul à 2,2GHz, 4Go de RAM). Une machine du cluster *helios* a été réservée pour l'exécution du super-nœud. Cette architecture présente un degré d'hétérogénéité *hd* de 0,5. Cette architecture est donc considérée comme neutre ;
- architecture 2.3 : elle est composée d'un total de 109 machines réparties sur 5 clusters de 4 sites. Les détails de sa composition sont : 35 machines du cluster *sol* à Sophia-Antipolis (4 cœurs de calcul à 2,6GHz, 4Go de RAM), 25 machines du cluster *chinqchint* à Lille (8 cœurs de calcul à 2,83GHz, 8Go de RAM), 20 machines du cluster *genepi* à Grenoble (8 cœurs de calcul à 2,5GHz, 8Go de RAM), et 30 machines du cluster *helios* à Sophia-Antipolis (4 cœurs de calcul à

2,2GHz, 4Go de RAM). Une machine du cluster *helios* a été réservée pour l'exécution du super-nœud. Cette architecture présente un degré d'hétérogénéité *hd* de 0,8. Cette architecture est donc considérée comme hétérogène.

Ainsi, les six architectures utilisées permettent de faire varier le degré d'hétérogénéité, ceci afin d'étudier l'efficacité des algorithmes de placement. Toutes ces architectures sont composées de plusieurs clusters répartis sur plusieurs sites, ce qui induit des latences dans les communications entre les tâches placées sur ceux-ci.

Pour la latence des liens de communication, qui est utilisée par l'algorithme de placement MAHEVE, nous avons dû utiliser des valeurs artificielles, celles-ci restant proches de la réalité. En effet, dans la bibliothèque « Mapping », il n'existe pas encore de mécanisme permettant de mesurer celle-ci avec précision. Afin de simplifier cette approche, il a été décidé d'utiliser les latences suivantes :

- latence entre deux machines au sein d'un même cluster : 0,10 ms ;
- latence entre deux machines d'un même site mais de deux clusters différents : 0,25 ms ;
- latence entre deux machines de sites différents : 10 ms.

Ces valeurs sont très proches des valeurs réelles que l'on peut mesurer sur la grille d'expérimentation Grid'5000.

5.1.2 Paramètrage des algorithmes de placement

Comme nous l'avons vu dans le chapitre précédent, chaque algorithme de placement possède divers paramètres :

- FT-FEC : le paramètre δ permet de déterminer combien de dépendances obligatoires doit avoir une tâche sur un même cluster – c'est-à-dire combien de dépendances de cette tâche doivent être placées sur le même cluster, afin de réduire les coûts de communication ;
- FT-AIAC-QM : le paramètre f, qui est un facteur de recherche, permet de déterminer le nombre de machines qui sont considérées pour la recherche d'une meilleure machine ;
- MAHEVE : le paramètre *nbSauv* détermine combien de machines doivent être réservées sur chaque cluster à des fins de tolérance aux pannes – ces machines réservées permettent de choisir des machines de remplacement dans le même cluster que celle qui vient de tomber en panne, ceci afin de préserver la localité des tâches et de réduire les coûts de transfert pour les mécanismes de tolérance aux pannes.

Pour les deux premiers paramètres, δ et f, nous avons mené des expériences, afin de pouvoir déterminer au mieux leur valeur. L'application que nous avons utilisée pour ces expériences est celle de la multidécomposition avec gradient conjugué présentée dans la section 5.1.3, avec une taille de matrice de 550000 de côté avec une largeur de bande de 35000, impliquant pour chacune des 64 tâches deux dépendances. L'architecture utilisée était composée de 122 machines, représentant 506 cœurs de calcul, réparties sur 5 clusters de 5 sites. Le degré d'hétérogénéité de cette architecture était de 0,5. Les deux paramètres ont successivement pris les valeurs 0,1, 0,5 et 0,9. Le tableau 5.1 donne les résultats des gains sur le temps d'exécution de l'application, en fonction de la variation de la valeur des paramètres.

Valeur du paramètre	0,1	0,5	0,9
f (FT-AIC-QM)	30%	32%	30%
δ (FT-FEC)	40%	37%	45%

TABLE 5.1 – Gains sur le temps d'exécution de l'application en fonction de la valeur des paramètres des algorithmes de placement

Tout d'abord, comme on peut le voir pour le paramètre f, la meilleure valeur se situe aux alentours de 0,5. Cependant, la différence de performances entre les valeurs n'est pas assez significative pour indiquer une configuration particulière. Concernant le paramètre δ, il n'est pas surprenant que sa meilleure valeur soit proche de 1. En effet, plus les tâches sont proches, plus les communications et la convergence sont rapides.

Ainsi, dans la suite de nos expériences, nous fixons $f = 0{,}5$ et $\delta = 0{,}9$. Pour le paramètre δ, nous ne l'avons pas fixé à 1 afin de laisser un certain degré de liberté à l'algorithme dans les choix de placement. Concernant le paramètre $nbSauv$ de MAHEVE, nous utilisons deux valeurs :

- $nbSauv = 0$ quand nous n'introduisons pas de panne pendant le calcul, permettant ainsi d'utiliser toutes les machines de chaque cluster ;
- $nbSauv = 2$ quand nous introduisons des pannes, permettant ainsi de réserver au moins deux machines sur chaque cluster pour la tolérance aux pannes.

Clairement, une étude plus approfondie mérite d'être menée afin d'évaluer au mieux ces paramètres. Néanmoins, le fait de se placer dans des conditions d'expérimentation réelles complexifie ce type d'étude.

5.1.3 Applications utilisées

Dans cette section nous décrivons les deux applications que nous avons utilisées pour la réalisation de nos expérimentations. La première, celle de la multidécomposition avec gradient conjugué, est une adaptation d'une application issue d'un ensemble d'applications de tests, le NAS NPB. La seconde est une application reposant sur la résolution de l'équation d'advection-diffusion en trois dimensions.

Multidécomposition avec gradient conjugué

Le programme NAS, pour « Numerical Aerodynamic Simulation » (simulation aérodynamique numérique), basé au centre de recherche Ames de la NASA, a développé un ensemble d'applications test, appelé « NAS Parallel Benchmark » [18]. Ces applications sont dédiées à l'évaluation des architectures à haute performance, comme par exemple les supercalculateurs hautement parallèles. Cet ensemble est constitué de cinq applications parallèles et de trois simulations.

L'application qui nous intéresse ici est la résolution de systèmes linéraires de la forme $Ax = b$ à l'aide de la méthode de la multidécomposition avec gradient conjugué. La parallélisation mise en œuvre ici utilise la méthode de la multidécomposition, décrite dans la suite de cette section. La résolution du problème utilise un solveur séquentiel au niveau de la tâche, réalisant le calcul du gradient conjugué sur des sous-ensembles du système linéaire. Cette méthode calcule une approximation de la plus petite valeur propre d'une matrice creuse de grande taille, symétrique à valeurs positives non nulles.

Nous décrivons ici l'association de ces deux méthodes, celle du gradient conjugué et celle de la multidécomposition pour résoudre des systèmes linéaires. La méthode itérative résultant de cette association utilise donc dans un premier temps la méthode de multidécomposition afin de découper le système linéaire en sous-systèmes, qui sont résolus dans un second temps séquentiellement par chaque tâche en utilisant une méthode classique du gradient conjugué. Cette association est compatible avec le modèle itératif asynchrone.

En utilisant la méthode de la multidécomposition [13], la matrice A est découpée en bande horizontales, comme le montre la figure 5.1. Chacune de ces bandes est affectée à une tâche de calcul. De ce fait, chaque tâche est en charge de calculer sa partie $XPart$ en résolvant le sous-système suivant : $APart \times XPart = BPart - DepGauche \times XGauche - DepDroite \times XDroite$. Dans le cadre de notre étude, la résolution de cette équation est effectuée par la méthode du gradient conjugué. À la fin d'une itération, la tâche envoie $XPart$, dont ont besoin ses dépendances pour leur calcul.

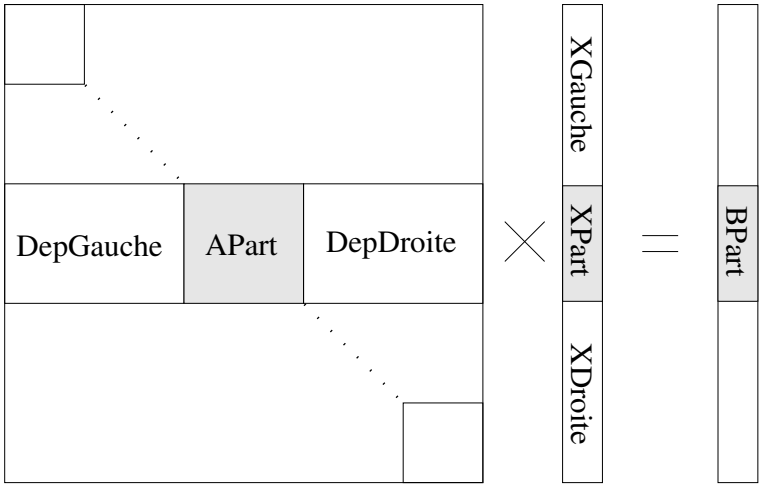

FIGURE 5.1 – Décomposition des données en utilisant la méthode de la multidécomposition

Cette méthode peut être décomposée en quatre phases :

1. **Décomposition des données.** Dans cette phase, les données sont allouées à chaque tâche suivant la décomposition illustrée par la figure 5.1. Ensuite, chaque tâche itère jusqu'à converger, en exécutant les phases suivantes ;
2. **Calcul.** Dans un premier temps, chaque tâche calcule $BLoc = BPart - DepGauche \times XGauche - DepDroite \times XDroite$. Ensuite, chacune résout $APart \times XPart = BLoc$ en utilisant une version séquentielle de la méthode du gradient conjugué ;
3. **Échange de données.** Chaque tâche envoie sa partie $XPart$ à ses dépendances. Ici, le nombre de dépendances est étroitement relié à la densité de la matrice A. En effet, une matrice dense implique un schéma de communications complet, où toutes les tâches sont interdépendantes, alors qu'une largeur de bande réduite diminue ce schéma de communications ;
4. **Détection de convergence.** Chaque tâche calcule sa convergence locale, et suivant le mécanisme de détection de la convergence globale, celle-ci peut être détectée de manière centralisée ou décentralisée.

Il est possible de modifier la décomposition des données pour obtenir des sous-ensembles non disjoints, afin de pouvoir appliquer le principe de recouvrement des communications par du calcul. Le fait d'utiliser des recouvrements permet d'accélérer la vitesse de convergence.

Résolution de l'équation d'advection-diffusion 3D

Le problème d'advection-diffusion représente mathématiquement les processus de transport de polluants, salinité ..., combinés avec leurs interactions biochimiques. Le système possède une valeur initiale et l'objectif ici est de déterminer son évolution au cours du temps. Ce problème se résout à l'aide d'équations différentielles partielles (EDP) non linéaires. Cette non linéarité provient des interactions biochimiques entre les différentes espèces. Plus de détails sur ce problème peuvent être trouvés dans [17].

Un système d'équations d'advection-diffusion-réaction a la forme suivante :

$$\frac{\partial c}{\partial t} + A(c, a) = D(c, d) + R(c, t) \tag{5.1}$$

dans laquelle c représente le vecteur des concentrations inconnues des espèces, de taille m, et les deux vecteurs

$$A(c,a) = [\mathbf{J}(c)] \times a^T, \quad (5.2)$$
$$D(c,d) = [\mathbf{J}(c)] \times d \times \nabla^T, \quad (5.3)$$

définissant respectivement les processus d'advection et de diffusion ($\mathbf{J}(c)$ représente la jacobienne de c). La vitesse de déplacement des fluides u, v et w dans le champ $a = (u, v, w)$, et la matrice des cœfficients de diffusion d sont supposés être définis. Par exemple, la simulation de la diffusion de polluants dans une eau peu profonde peut être réalisée si a est obtenu à l'aide d'un modèle hydrodynamique. Le transport d'espèces chimiques est défini par les processus d'advection et de diffusion, en considérant R comme les réactions, émissions, ou absorptions des interactions entre les espèces.

De plus amples informations concernant la description mathématique de ce problème peuvent être trouvées dans [66]. De même, concernant sa mise œuvre, des informations plus détaillées sont disponibles dans [39].

La parallélisation de la résolution de l'équation d'advection-diffusion tridimensionnelle est réalisée ici à l'aide de la méthode de la multidécomposition, présentée précédemment. L'usage de cette méthode de parallélisation permet d'utiliser cette application avec le modèle itératif asynchrone.

5.2 Résultats

5.2.1 Multidécomposition avec gradient conjugué

Dans un premier temps nous donnons les résultats des expériences utilisant l'application de la multidécomposition avec gradient conjugué. La taille des matrices utilisées ici est de 5000000 de côté, avec une largeur de bande de 35000. Ces paramètres engendrent en moyenne 2 dépendances pour chaque tâche de calcul. Le découpage des données est fait de telle sorte que nous obtenons 64 tâches de calcul.

Les architectures utilisées sont celles décrites dans la section 5.1.1, soit les architectures 1.1, 1.2, et 1.3, de degrés d'hétérogénéité respectifs de 0,08, 0,5, et 0,72.

Le protocole d'expérimentation que nous avons utilisé est le suivant : pour chaque architecture d'exécution, nous avons effectué plusieurs exécutions de l'application, afin de déterminer un temps moyen d'exécution. De plus, nous avons mis en place deux types d'exécution : des exécutions sans panne de machine, et d'autres avec 2 pannes toutes les 20 secondes. Les machines sélectionnées pour ces pannes sont choisies au hasard, exclusivement parmi celles impliquées dans le calcul – c'est-à-dire qu'aucune machine disponible n'a été affectée par une panne au cours de ces expériences. Il est à noter que les machines ayant subi une mise en panne volontaire au cours des calculs rejoignent de nouveau l'architecture après un laps de temps prédéfini de 30 secondes.

Les tableaux 5.2 et 5.3 présentent les résultats obtenus, en donnant les temps moyens d'exécution de l'application en secondes, ainsi que les gains par rapport aux exécutions sans algorithme de placement.

Comme on peut le remarquer, tous les algorithmes de placement permettent de réduire le temps d'exécution de l'application. Concernant nos deux algorithmes classiques FT-FEC et FT-AIAC-QM, on remarque que la meilleure efficacité de chacun d'eux se porte sur un type d'architecture : les architectures homogènes (*hd* proche de 0) pour le premier, et les architectures hétérogènes (*hd* supérieur à 0,5) pour le second. Ces résultats correspondent à nos attentes sur leur comportement. Ici, les gains obtenus montent jusqu'à 23,7% pour FT-FEC et jusqu'à 22% pour FT-AIAC-QM.

hd \ Algorithme		Sans	FT-AIAC-QM	FT-FEC	MAHEVE
0,08	temps	80s	63s	61s	60s
	gains	–	22%	23,7%	24,9%
0,5	temps	67s	61s	62s	54s
	gains	–	8,5%	6,5%	18,5%
0,72	temps	67s	59s	66s	52s
	gains	–	11,9%	2%	21,9%

TABLE 5.2 – Temps d'exécution et gains pour l'application de la multidécomposition avec gradient conjugué, sans panne

Concernant notre algorithme hybride, MAHEVE, on peut remarquer qu'il permet d'obtenir des gains plus importants que les deux autres algorithmes, et ce pour toutes les architectures. Ceci montre bien que cet algorithme s'adapte en fonction du degré d'hétérogénéité de l'architecture d'exécution. Non seulement il adopte un comportement similaire au meilleur des deux autres algorithmes suivant le degré d'hétérogénéité, mais en plus sa prise en compte des caractéristiques de l'application le rend plus performant. Les gains obtenus montent jusqu'à 24,9%.

hd \ Algorithme		Sans	FT-AIAC-QM	FT-FEC	MAHEVE
0,08	temps	229s	179s	154s	113s
	gains	–	22,1%	32,8%	50,7%
0,5	temps	242s	118s	134s	85s
	gains	–	51,2%	44,8%	64,9%
0,72	temps	193s	100s	122s	86s
	gains	–	48,3%	36,9%	55,3%

TABLE 5.3 – Temps d'exécution et gains pour l'application de la multidécomposition avec gradient conjugué, avec 2 pannes chaque 20 secondes

Pour les expériences avec des pannes de machines durant l'exécution de l'application, les résultats précédents sont confirmés. Tout d'abord, tous les algorithmes permettent d'abaisser le temps d'exécution de l'application, mais on peut remarquer ici que les gains sont plus importants. Pour FT-FEC et FT-AIAC-QM, les gains maximum obtenus pour chacun d'eux sont respectivement de 44,8% et 51,2%.

Concernant notre algorithme MAHEVE, il est encore une fois plus performant que les deux autres, en permettant des gains sur le temps d'exécution de l'application allant jusqu'à 64,9%, ce qui est 13,7 points de mieux que le meilleur des deux autres algorithmes. Le facteur permettant d'expliquer cet écart de performances, venant s'ajouter à ceux cités précédemment, est l'intégration d'une politique de tolérance aux pannes dans le placement initial. On peut voir ici que le fait de réserver quelques machines sur les clusters durant cette phase permet d'obtenir de meilleures performances lorsque des pannes surviennent au cours de l'exécution de l'application.

Le tableau 5.4 présente les ratios de l'impact des pannes sur le temps d'exécution de l'application. Ce ratio est calculé en divisant le temps d'exécution de l'application avec des pannes, par celui des

exécutions sans panne. Ce ratio nous donne une indication sur l'efficacité des politiques de tolérance aux pannes de nos algorithmes.

hd \ Algorithme	Sans	FT-AIAC-QM	FT-FEC	MAHEVE
0.08	2,9	2,8	2,5	1,9
0.5	3,6	1,9	2,1	1,6
0.72	2,9	1,7	1,9	1,6

TABLE 5.4 – Ratios d'impact des pannes sur le temps d'exécution de l'application de la multidécomposition avec gradient conjugué

Comme on peut le voir, les politiques de tolérance aux pannes mises en place dans nos algorithmes de placement permettent de réduire l'impact des pannes sur le temps d'exécution de l'application. Pour nos deux algorithmes classiques, FT-FEC et FT-AIAC-QM, on remarque qu'ils procurent une meilleure réduction de cet impact pour leurs architectures cibles respectives. Pour FT-FEC, le meilleur ratio est de 1,9, et pour FT-AIAC-QM le meilleur ratio est de 1,7, ce qui est respectivement 1 et 1,2 de mieux par rapport aux exécutions sans algorithme de placement. Ceci indique que les politiques de tolérance aux pannes des deux algorithmes sont plutôt bonnes.

Concernant notre algorithme MAHEVE, il est meilleur que les deux autres, en procurant tout le temps le ratio le plus faible. Le meilleur ratio de cet algorithme est de 1,6, ce qui est 1,3 de mieux par rapport aux exécutions sans algorithme de placement.

Un autre point important à noter ici, concernant le modèle itératif asynchrone, est le fait que les temps d'exécution pour une même application sont plus petits sur des architectures hétérogènes. Ceci provient aussi du fait que certaines machines présentes dans ces architectures sont bien plus puissantes (8 cœurs de calcul) que celles présentes dans les architectures homogènes.

5.2.2 Résolution de l'équation d'advection-diffusion 3D

Dans cette section nous présentons les résultats des expériences réalisées avec l'application de la résolution de l'équation d'advection-diffusion. Les paramètres donnés à l'application sont : 150 pour la taille des côtés de la grille tridimensionnelle, et 2 pour le nombre d'espèces chimiques en jeu. Nous avons choisi de paralléliser cette application en 64 tâches. Avec ce paramétrage, chaque tâche possède en moyenne 8 dépendances de calcul.

Les architectures utilisées ici sont celles décrites dans la section 5.1.1, soit les architectures 2.1, 2.2, et 2.3, de degrés d'hétérogénéité respectifs 0,05, 0,5, et 0,8.

Le protocole d'expérimentation utilisé est le même que pour l'application de la multidécomposition avec gradient conjugué, à savoir de multiples exécutions de l'application sur chacune des architectures afin de déterminer un temps moyen d'exécution, et avec deux séries d'expériences : l'une sans panne et l'autre avec 2 pannes toutes les 20 secondes.

Les tableaux 5.5 et 5.6 présentent les résultats obtenus, en donnant les temps moyens d'exécution de l'application en secondes, ainsi que les gains par rapport aux exécutions sans algorithme de placement.

On remarque que dans ces expériences, tout comme dans celles avec l'application de la multidécomposition avec gradient conjugué, tous les algorithmes de placement permettent de réduire les temps d'exécution de l'application. Les gains sont cependant moins importants. Ceci peut s'expliquer par le fait que les tâches de cette application ont plus de dépendances et aussi un volume de communication,

hd \ Algorithme		Sans	FT-AIAC-QM	FT-FEC	MAHEVE
0,05	temps	326s	307s	304s	298s
	gains	–	5,8%	6,7%	8,6%
0,5	temps	335s	307s	302s	291s
	gains	–	8,4%	9,9%	13,1%
0,8	temps	328s	302s	305s	293s
	gains	–	7,9%	7%	10,7%

TABLE 5.5 – Temps d'exécution et gains pour l'application de la résolution de l'équation d'advection-diffusion 3D, sans panne

pour chaque dépendance, plus élevé. De même, comme la charge de calcul des tâches varie au cours du temps, le placement peut s'avérer un bon choix au début de l'exécution, mais devenir moins bon au cours du temps. Malgré tout, FT-FEC permet de réduire au mieux de 9,9% le temps d'exécution, et FT-AIAC-QM le réduit de 8,4%. On peut remarquer que FT-FEC devient plus performant que FT-AIAC-QM pour une architecture de degré d'hétérogénéité 0,5, contrairement aux résultats de l'application précédente. Ceci provient du fait que les tâches possèdent plus de dépendances et ont plus de communications, impliquant alors un plus fort impact de la localité des tâches sur le temps d'exécution de l'application.

On peut ici également remarquer que notre algorithme MAHEVE permet encore une fois de réduire au plus le temps d'exécution de l'application. En effet, il permet de le réduire jusqu'à 13,1%, ce qui est 3,2 points de plus que FT-FEC et 4,7 points de plus que FT-AIAC-QM.

hd \ Algorithme		Sans	FT-AIAC-QM	FT-FEC	MAHEVE
0,05	temps	459s	421s	420s	394s
	gains	–	8,2%	8,6%	14,3%
0,5	temps	487s	433s	425s	408s
	gains	–	11,1%	12,7%	16,2%
0,8	temps	454s	430s	435s	397s
	gains	–	5,3%	4,2%	12,6%

TABLE 5.6 – Temps d'exécution et gains pour l'application de la résolution de l'équation d'advection-diffusion 3D, avec 2 pannes chaque 20 secondes

Concernant les expériences avec des pannes de machines en cours de calcul, on peut remarquer le même comportement que pour l'application de la multidécomposition avec gradient conjugué. Ici, FT-FEC permet de réduire le temps d'exécution de l'application jusqu'à 12,7% et FT-AIAC-QM jusqu'à 11,1%. On remarque également que FT-FEC est plus performant sur l'architecture de degré d'hétérogénéité 0,5, toujours en conséquence de la prédominance de la localité des tâches.

Concernant notre algorithme hybride MAHEVE, on remarque qu'ici aussi il est plus performant que les deux autres. Il permet d'obtenir une diminution des temps d'exécution de l'application jusqu'à

16,2%, ce qui est 3,5, points de plus que FT-FEC et 5,1 points de plus que FT-AIAC-QM. Ces plus grands écarts, par rapport aux exécutions sans panne, proviennent du fait que les machines réservées sur les clusters, en prévision de la tolérance aux pannes, permettent de remplacer des machines défaillantes par d'autres très proches. Ceci permet ainsi de conserver la localité des tâches de calcul, qui comme on l'a vu précédemment est très importante pour cette application.

Le tableau 5.7 présente les ratios de l'impact des pannes sur le temps d'exécution de l'application.

Algorithme / *hd*	Sans	FT-AIAC-QM	FT-FEC	MAHEVE
0,05	1,4	1,4	1,4	1,3
0,5	1,4	1,4	1,3	1,2
0,8	1,4	1,4	1,4	1,4

TABLE 5.7 – Ratios d'impact des pannes sur le temps d'exécution de l'application de la résolution de l'équation d'advection-diffusion 3D

Pour cette expérimentation, les gains apportés à la réduction de l'impact des pannes de machines sur le temps d'exécution de l'application sont bien moins importants que pour la précédente. Pour FT-AIAC-QM, il n'y a aucun apport, par rapport aux exécutions sans algorithmes de placement, ce qui indique que pour ce genre d'application sa politique de tolérance aux pannes n'est pas efficace. Concernant FT-FEC, il prend un léger avantage par rapport à FT-AIAC-QM sur l'architecture de degré d'hétérogénéité 0,5.

Concernant notre algorithme hybride MAHEVE, on peut remarquer qu'il permet de réduire ce ratio jusqu'à 1,2, avec une différence de 0,2 par rapport aux exécutions sans algorithme de placement, ce qui est moins bien que pour l'application de multidécomposition avec gradient conjugué, mais qui est mieux que les deux autres algorithmes de placement. Ceci permet de dire que MAHEVE a une meilleure politique de tolérance aux pannes que les deux autres algorithmes, mais que celle-ci pourrait encore être améliorée.

5.3 Conclusion

Dans ce chapitre nous avons présenté les expérimentations que nous avons réalisées afin d'évaluer les performances de nos algorithmes de placement pour l'exécution d'applications itératives asynchrones sur des environnements hétérogènes et volatils.

Dans un premier temps nous avons présenté les conditions de nos expérimentations. Tout d'abord nous avons décrit les plateformes utilisées, qui sont des sous-ensembles de la grille d'expérimentation Grid'5000. Chaque architecture offre plus de 100 machines de calcul pour nos expériences.

Ensuite nous avons présenté les deux applications que nous avons utilisées. La première, la multidécomposition avec gradient conjugué, est une application typique du modèle itératif asynchrone. Les données que nous avons utilisées pour les exécutions de cette application ont généré en moyenne 2 dépendances par tâche de calcul. La seconde application, la résolution de l'équation d'advection-diffusion en trois dimensions, permet quant à elle de modéliser des phénomènes naturels de propagation, comme par exemple celle de polluants dans un milieu aquatique. Le modèle utilisé ici repose sur des équations différentielles partielles, qui en fonction des données que nous avons utilisées et avec l'aide de transformations peuvent être résolues comme un système linéaire. Ces deux applications ont été paramétrées afin de pouvoir obtenir 64 tâches de calcul.

Dans un second temps, nous avons présenté les résultats de nos expérimentations. Chaque application a été exécutée plusieurs fois sur chaque architecture, afin de pouvoir obtenir un temps moyen d'exécution de chaque application. Ces exécutions ont été faites suivant deux modes opératoires : le premier n'incluant pas de panne pendant l'exécution de l'application, le second en incluant 2 toutes les 20 secondes. Les machines sélectionnées pour la mise en œuvre des pannes artificielles ont été choisies aléatoirement parmi celles participant au calcul, et non celles restant disponibles.

Les résultats de ces expériences ont tout d'abord confirmé le fait que l'utilisation d'un algorithme de placement permet de réduire considérablement le temps d'exécution d'une application itérative asynchrone. Nous avons également pu vérifier que les deux algorithmes, FT-FEC et FT-AIAC-QM, sont plus ou moins efficaces selon l'architecture cible. Nous avons montré que notre algorithme hybride de placement, MAHEVE, est meilleur que les deux autres algorithmes, et ce sur tous les types d'architectures testés. Nous avons pu remarquer cependant que lorsque les tâches d'une application possèdent de nombreuses dépendances et que celles-ci échangent un volume important de données, les algorithmes de placement doivent fortement en tenir compte. Les résultats observés indiquent qu'une étude plus approfondie pour la prise en compte des caractéristiques des applications doit être menée afin d'améliorer les stratégies de placement. De même, nous avons pu voir que pour une application dont la charge de calcul des tâches varie en cours d'exécution, un simple placement initial ne suffit pas ; il faudrait alors utiliser des méthodes d'équilibrage de charge dynamique.

En utilisant des expériences avec des pannes au cours des exécutions, nous avons pu mesurer l'efficacité des politiques de tolérance aux pannes des algorithmes. Nous avons pu voir que les politiques simples des algorithmes FT-FEC et FT-AIAC-QM atteignent assez rapidement leurs limites. En revanche, pour notre algorithme hybride MAHEVE, qui inclut dans sa conception des mécanismes pour la tolérance aux pannes, sa politique s'avère bien meilleure. Ceci est d'autant plus vrai lorsque les tâches possèdent beaucoup de dépendances et que celles-ci échangent un important volume de données. Néanmoins, bien que plus performante que celles des deux autres algorithmes, la politique de tolérance aux pannes de MAHEVE peut encore être améliorée. Les améliorations possibles sont décrites dans la section perspectives de la conclusion générale.

Troisième partie

Virtualisation

Dans cette troisième et dernière partie, nous présentons notre second axe de recherche. Celui-ci porte sur la mise en œuvre d'une plateforme de calcul pour les environnements de type « enterprise desktop grid », en utilisant des machines virtuelles.

Les « enterprise desktop grids » sont des architectures de calcul regroupant les machines présentes au sein d'institutions, qu'elles soient des entreprises ou des entités publiques comme des universités ou des laboratoires de recherche. Les machines utilisées sont des machines de travail et non des machines dédiées au calcul comme peuvent l'être des clusters par exemple. Le principe ici est d'utiliser le potentiel non exploité de ces machines et surtout leur temps de non utilisation, car ces machines sont de plus en plus puissantes et offrent beaucoup de ressources, comme la mémoire vive et l'espace de stockage, mais également des unités de calcul de plus en plus puissantes.

En effet, de nombreuses personnes voulant exécuter des applications de calcul, comme des scientifiques ou des industriels, ne peuvent pas avoir accès à des plateformes de calcul comme les grilles de calcul. Leurs applications devant traiter des informations volumineuses, leurs temps d'exécution sont très longs (plusieurs jours). De plus, les données et processus utilisés sont souvent confidentiels. Ceci empêche l'utilisation de solutions comme le « cloud computing », dont le principe est d'exécuter les applications sur une architecture distribuée, gérée par une entreprise externe. De ce fait, la confidentialité des données et des processus peut être compromise. Ainsi, nous proposons la mise en œuvre d'une plateforme de calcul, exécutée en interne de l'institution (pour préserver la confidentialité des données et des processus) et utilisant des machines virtuelles.

L'utilisation de machines virtuelles permet une gestion plus souple de la plateforme, et offre un environnement d'exécution plus sécurisé pour les applications et leurs données. En effet, suivant la technique de virtualisation utilisée, l'isolation des applications exécutées dans une machine virtuelle est plus ou moins forte. Seules deux techniques permettent une isolation forte des applications et de leurs données dans une machine virtuelle. Ces techniques utilisent un hyperviseur, qui est en charge de la gestion des machines virtuelles et de leur liaison avec le matériel de la machine hôte. Chacune de ces techniques apportent des avantages et des inconvénients, et il revient à l'utilisateur de déterminer quels sont ses besoins et quelles sont ses possibilités de mise œuvre de ces solutions.

Dans cette partie nous présentons tout d'abord dans le chapitre 6 les motivations de la mise en œuvre d'une plateforme de calcul utilisant des machines virtuelles dans un environnement de type « enterprise desktop grid ». Ensuite nous décrivons les concepts de la virtualisation ainsi que ses enjeux majeurs. Puis nous décrivons les principales techniques de virtualisation existantes, en donnant leurs avantages et leurs inconvénients. Enfin nous présentons deux plateformes de calcul utilisant des machines virtuelles, chacune utilisant une technique de virtualisation différente.

Le chapitre 7 présente notre plateforme de calcul, HpcVm. Elle utilise des machines virtuelles pour exécuter des applications de calcul dans des environnements de type « enterprise desktop grid ». Dans un premier temps, nous présentons les choix techniques que nous avons effectués pour sa mise en œuvre. Ensuite nous décrivons l'architecture de la plateforme et nous détaillons son fonctionnement. Enfin, nous présentons les résultats des expériences que nous avons menées afin d'évaluer ses performances. Nous y décrivons les résultats des exécutions de deux applications de calcul, en comparant différents types d'exécution.

Chapitre 6

État de l'art

Dans ce chapitre nous présentons les différents aspects de l'utilisation des techniques de virtualisation pour effectuer des calculs scientifiques. Dans un premier temps nous présentons les motivations de l'utilisation de ces techniques pour la création de plateformes de calcul. Puis nous définissions le concept de la virtualisation et quels en sont les enjeux. Ensuite sont présentées les différentes techniques de virtualisation. Chacune possède des avantages tant en terme de performances qu'en terme de facilité d'utilisation, mais bien souvent ces deux avantages ne sont pas compatibles. Enfin, dans une dernière section nous présentons deux plateformes de calcul utilisant des machines virtuelles.

6.1 Motivations

Nous sommes partis du constat simple à la base du calcul volontaire : dans la majorité des institutions, qu'elles soient des entreprises ou des entités publiques comme des universités, le potentiel de beaucoup de machines n'est pas totalement exploité, loin s'en faut. Bien souvent, pour des raisons de versions de logiciels, les machines sont remplacées par des machines plus récentes, sans que pour autant l'utilisation des ressources soit modifiée. Ainsi, des machines puissantes, comportant aujourd'hui plusieurs cœurs de calcul cadencés à de hautes fréquences, avec une grande quantité de mémoire (au moins 4Go), et un grand espace de stockage, servent toujours pour des besoins basiques, qui n'ont pas spécialement évolués. De ce fait, les machines sont sous-exploitées. C'est pourquoi nous avons décidé de créer une plateforme de calcul utilisant ces ressources disponibles. Cette plateforme utilise des machines virtuelles, afin de s'affranchir des dépendances vis-à-vis du système d'exploitation hôte et du matériel présent, surtout dans un environnement où ceux-ci sont souvent hétérogènes

Il existe multiples intérêts liés à l'utilisation de machines virtuelles dans un contexte de calcul hautes performances. On peut citer les suivants :

- utilisation optimale des ressources disponibles. Les machines virtuelles peuvent être déployées sur toutes les machines inoccupées, rentabilisant ainsi leur fonctionnement. De plus, si une machine possède trop de charge de calcul, les machines virtuelles s'exécutant sur celle-ci peuvent être assez facilement déplacées sur une autre moins chargée, et ainsi répartir la charge de la plateforme ;
- installation et déploiement aisées d'applications de calcul. Avec l'utilisation des machines virtuelles, il n'est plus nécessaire d'effectuer l'installation des applications et de leurs environnements d'exécution, ainsi que leur configuration, sur de multiples machines, mais seulement à un

seul endroit : l'image de la machine virtuelle. Ensuite il suffit de déployer cette image sur toutes les machines hôtes de la plateforme. De ce fait, toute la configuration devient indépendante des mises à jour de l'hôte, qui dans bien des cas entraînent une phase de reconfiguration des environnements pour l'exécution des applications. De même, lorsque de nouvelles machines viennent rejoindre la plateforme, il suffit d'installer les composants de gestion de la plateforme pour qu'elle soit opérationnelle ;
- migration des calculs facilitée. Lorsqu'une machine fait face à une trop forte charge, provenant des machines virtuelles ou bien de son utilisation normale, les machines virtuelles, ainsi que les applications qu'elles exécutent, peuvent migrer plus facilement sur une autre machine. Les applications sont moins impactées par ce genre d'opération que si elles étaient exécutées directement dans le système hôte ;
- économie de matériel et d'investissement. Le principe d'une telle plateforme est d'utiliser les ressources disponibles qui sont sous-exploitées. Il n'est donc nul besoin d'investir dans une architecture dédiée, comme un cluster par exemple. Il est bien sûr évident qu'une architecture dédiée permet d'obtenir une plus haute disponibilité des machines et bien souvent une puissance plus élevée, mais avec un coût supplémentaire non négligeable. De même, dans bien des cas, les machines de travail restent en fonctionnement durant les heures de pause et la nuit. L'utilisation d'une plateforme comme celle que nous proposons permet de rentabiliser ces coûts de fonctionnement ;
- sécurité et isolation des calculs. Grâce à l'utilisation de machines virtuelles, les applications exécutées en leur sein sont isolées du reste de la machine, donc de toute interférence néfaste. Il convient de nuancer cette sécurité au regard de la technique de virtualisation employée et aussi des compétences des attaquants (la sécurité à 100% n'existant pas dans un tel contexte). En effet, il est difficile pour une personne n'ayant pas accès à la machine virtuelle de perturber les calculs s'y déroulant. L'intégrité des calculs est ainsi préservée. De même, la protection peut être retournée, c'est-à-dire que la machine hôte, et par extension ses utilisateurs, est elle aussi protégée. Il peut arriver qu'une application soit mal programmée, ou bien qu'elle contienne un code malveillant. Si une telle application est exécutée dans une machine virtuelle, alors les dégâts provoqués n'impacteront que la machine virtuelle et non l'hôte. Ainsi les utilisateurs « normaux » peuvent continuer de travailler normalement, et la machine virtuelle « abîmée » doit simplement être remplacée ;
- préservation de la confidentialité des données. L'utilisation d'une solution de « cloud computing » implique que les applications mais surtout leurs données et résultats soient externalisés. De ce fait la confidentialité des données peut être remise en cause. Avec l'utilisation d'une plateforme interne à l'institution, cette confidentialité est préservée ;
- tolérance aux pannes. Le fait d'utiliser des machines virtuelles pour effectuer les calculs permet de faciliter la mise en place de mécanismes de tolérance aux pannes. En effet, il n'est plus nécessaire d'intégrer une politique de gestion des pannes au sein de l'application de calcul, mais seulement un appel à une fonction demandant à la plateforme de déclencher un tel mécanisme. Une solution simple est de sauvegarder l'état des machines virtuelles, et lorsqu'une panne survient, il suffit de déployer la dernière sauvegarde de la machine virtuelle devenue indisponible.

Notre objectif est de permettre l'exécution d'applications de calcul sur des architectures de type « enterprise desktop grid ». Les utilisateurs ciblés pour l'usage de notre plateforme sont des universitaires ou des industriels dont les applications requièrent de longs temps de calcul (plusieurs heures, voire plusieurs jours). De même, ces personnes n'ont pas le temps ni forcément les compétences pour adapter leurs programmes afin de gérer les divers aspects intervenant dans une plateforme d'exécution distribuée, comme la gestion des environnements d'exécution (recompilation et choix des exécutables en fonction de l'architecture de la machine) et la gestion des pannes (intégrer des mécanismes de tolérance aux pannes). Ainsi, le but de notre plateforme est de permettre l'exécution d'applications de calcul

en utilisant les ressources disponibles au sein d'une entité, tout en étant la plus transparente possible pour les utilisateurs non informaticiens. De même, la mise en place de la plateforme doit être la plus simple possible, et ne doit pas entraîner des installations et des configurations longues et complexes. Un autre point important est que l'exécution des applications doit requérir le moins de modifications possible dans leur code, facilitant ainsi l'utilisation de la plateforme. Enfin, comme indiqué dans [43], les mécanismes de tolérance aux pannes est très important pour les applications s'exécutant sur de longues durées, car la probabilité d'apparition de pannes augmente exponentiellement ainsi que leur coût.

6.2 Principes de la virtualisation

L'idée même de simuler plusieurs machines sur une seule peut sembler a priori étrange. En effet, sachant qu'un système d'exploitation est conçu pour tirer parti du meilleur du matériel, le fait d'utiliser plusieurs systèmes engendre des interrogations quant à l'efficacité finale d'un tel usage. L'utilisation de plusieurs systèmes d'exploitation, qui ne sont pas prévus pour communiquer entre eux, du moins lorsqu'ils sont sur une même machine, peut entraîner des pertes de performances. Ces pertes se retrouvent au niveau des accès au disque dur, au niveau de l'utilisation du/des processeurs, et aussi au niveau de l'utilisation de la mémoire. Ainsi, le processus de virtualisation entraîne une consommation supplémentaire de ressources.

Cependant, pour chacun de ces problèmes, des solutions peuvent être mises en place afin de limiter les pertes de performance. Premièrement, concernant les temps d'accès au disque dur, plusieurs disques physiques peuvent être mis à la disposition des différents systèmes d'exploitation. La technique la plus efficace est de dédier un disque pour chacun d'eux, réduisant par conséquent les conflits lors des opérations de lecture et d'écriture sur ceux-ci. Concernant, les processeurs, les machines actuelles proposent en standard soit plusieurs processeurs, soit plusieurs cœurs de calcul sur un processeur, voire une combinaison des deux. Ainsi, chaque machine virtuelle peut obtenir l'utilisation d'un processeur ou d'un cœur de calcul qui lui est dédié. Ce mécanisme permet lui aussi d'éviter la gestion des demandes multiples pour une même ressource. Enfin, concernant la mémoire centrale, les machines actuelles proposent de plus en plus de mémoire disponible – il est à noter que les programmes actuels en consomment également de plus en plus, mais la quantité de mémoire disponible reste assez élevée. Celle-ci peut être partagée entre tous les systèmes, sans que ceux-ci ressentent une limitation particulière.

Différentes notions sont liées aux techniques de virtualisation. Ces notions sont les suivantes :

- système d'exploitation hôte : cette notion représente le système d'exploitation qui est installé directement sur le matériel, comme sur une machine classique ;
- partitionnement, isolation et/ou partage de ressources : cette notion représente le fait de partager les ressources disponibles en les cloisonnant au sein d'un système d'exploitation ;
- couche d'abstraction matérielle/logicielle : cette notion représente un mécanisme intermédiaire entre la machine physique ou l'hôte et la machine virtuelle, permettant à celle-ci d'utiliser des composants standardisés (ceci permet de s'affranchir des spécificités matériel des machines physiques) ;
- système d'exploitation virtualisé ou invité : cette notion représente un système d'exploitation qui s'exécute au sein d'un système hôte (que ce soit un système d'exploitation ou un gestionnaire de virtualisation) ;
- images manipulables : cette notion représente le fait que les images des machines virtuelles peuvent subir des opérations, telles que le démarrage, l'arrêt, la mise en veille, la pause, la sauvegarde et la restauration, ainsi que des opérations de déplacement ;

– réseau virtuel : cette notion représente l'utilisation d'un réseau spécifique à la virtualisation, en utilisant un réseau logiciel, un réseau entre les machines virtuelles au sein de l'hôte, ou entre l'hôte et les machines virtuelles, voire entre les machines virtuelles.

Chaque technique de virtualisation utilise une ou plusieurs de ces notions.

6.3 Techniques de virtualisation

Dans cette section nous présentons les solutions de virtualisation mettant en œuvre les notions présentées dans la section précédente. Ces techniques sont présentées dans l'ordre des moins invasives, c'est-à-dire celles nécessitant peu d'intervention de la part de l'administrateur de la machine hôte, à celles nécessitant une mise en œuvre plus complexe. Ainsi, nous descendons dans les couches d'empilement de la structure d'une machine. Comme nous le montrons dans les sous-sections suivantes, plus la solution utilisée se situe à un niveau bas, plus les performances des machines virtuelles augmentent.

6.3.1 Isolateur

Un isolateur est un mécanisme logiciel permettant d'isoler l'exécution d'un programme, ou dans le cadre de cette thèse les tâches d'une application, du reste des applications s'exécutant au sein du système d'exploitation hôte. Pour ce faire, le programme est exécuté dans un contexte dédié, appelé aussi zone d'exécution, dans lequel il est le seul à pouvoir agir. Aucun autre programme ne peut intervenir dans sa zone mémoire, mais les ressources restent partagées comme pour un programme ordinaire. Il est important de noter que la sécurité proposée par cette solution est plus élevée que pour un programme classique, mais les risques d'attaque restent présents. Ce mécanisme permet d'exécuter plusieurs instances d'une même application, même lorsque celle-ci n'a pas été prévue à cet effet.

La figure 6.1 présente l'architecture d'utilisation d'un isolateur.

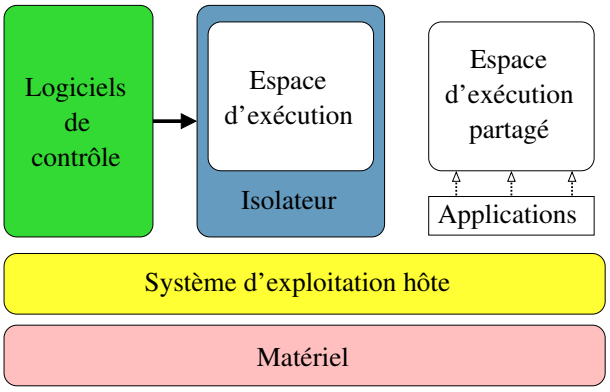

FIGURE 6.1 – Principe de fonctionnement d'un isolateur

Comme on peut le remarquer, ce mécanisme d'isolation est exécuté au sein d'un système d'exploitation hôte, qui est installé directement sur le matériel. Afin de gérer les applications isolées, des logiciels de contrôle sont utilisés. Cette solution permet de réduire les temps de non activité de la machine en exécutant plusieurs applications. Cette approche permet d'obtenir de bonnes performances. Son principal inconvénient est que les applications ne sont pas totalement isolées. En effet, elles utilisent le même espace de stockage, car seul l'environnement d'exécution est protégé. On ne parle pas ici de machine virtuelle, mais d'exécution virtualisée.

Bien que ce mécanisme permette d'isoler l'exécution des applications, ces exécutions restent classiques. De ce fait, il est difficile d'utiliser des mécanismes permettant la sauvegarde et la restauration des exécutions, et encore moins de pouvoir les déplacer sur une autre machine.

De plus, ces solutions ne sont actuellement disponibles que pour les systèmes d'exploitation utilisant des noyaux Unix/Linux. Les isolations peuvent être créées à l'aide par exemple des composants suivants : *Linux-VServer* (isolation des processus en espace utilisateur), *chroot* (isolation par changement de la racine du système), *BSD Jail* (isolation en espace utilisateur), ou *OpenVZ* (cloisonnement au niveau du noyau).

6.3.2 Noyau en espace utilisateur

Le principe de cette technique est d'exécuter un noyau de système d'exploitation dans l'espace utilisateur, tout comme le serait une application classique, dans un système hôte. Ici, ce noyau en espace utilisateur possède son propre espace utilisateur, différent de celui de la machine hôte et permet donc de cloisonner l'exécution des applications.

La figure 6.2 présente le principe d'utilisation de l'isolation à l'aide de l'exécution d'un noyau en espace utilisateur.

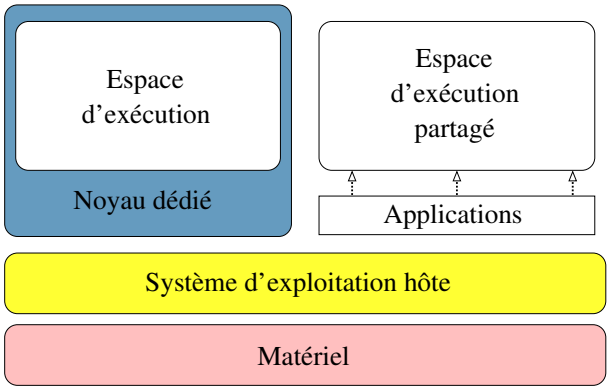

FIGURE 6.2 – Principe de fonctionnement de l'utilisation d'un noyau en espace utilisateur

Comme on peut le remarquer, cette technique met en place un noyau dédié pour chaque application. Ce noyau est donc exécuté comme une application par le système hôte. Chaque noyau dédié exécute alors une application. Cette technique permet une isolation de la gestion de l'application et de son espace mémoire. Cependant, cette technique n'assure pas une isolation totale de l'application, ni l'indépendance par rapport au matériel.

Encore une fois, ces solutions ne sont disponibles essentiellement que pour les systèmes d'exploitation utilisant un noyau Unix/Linux. Les outils utilisés pour la mise en œuvre d'une telle solution sont par exemple : *User Mode Linux* [26] (noyau s'exécutant en espace utilisateur), *Adeos* et *L4Linux* (micro noyaux temps réel), et *coLinux* (noyau pouvant s'exécuter sur un hôte Windows).

6.3.3 Hyperviseur de type 2

Un hyperviseur est un logiciel de virtualisation servant de plateforme permettant l'exécution de plusieurs systèmes d'exploitation sur un même hôte. Ici cet hyperviseur est un logiciel exécuté par le système hôte, comme une application classique.

La figure 6.3 présente le principe d'utilisation d'un hyperviseur de type 2. Comme on peut le voir sur la figure, l'hyperviseur de type 2 se décompose en deux parties :

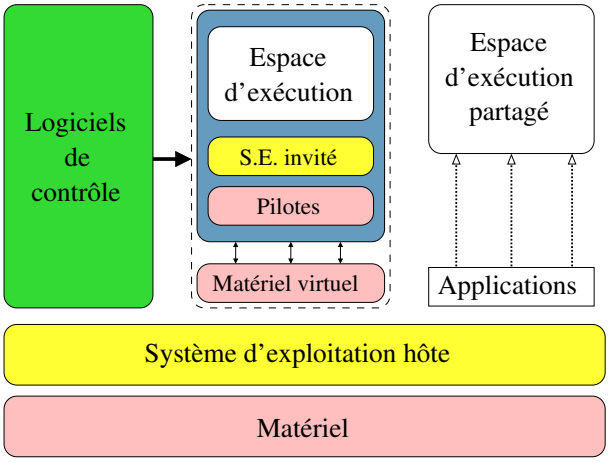

FIGURE 6.3 – Principe de fonctionnement d'un hyperviseur de type 2

- la virtualisation du matériel : cette partie permet de s'affranchir de la dépendance au matériel. Le but de cette virtualisation est de fournir aux systèmes invités toujours le même « matériel », et ce quel que soit le matériel réel sous-jacent. Ceci procure l'avantage qu'une machine virtuelle peut être exécutée sur n'importe quelle machine physique, dans la mesure où l'hyperviseur peut y être installé. Ainsi, il est offert la possibilité de n'utiliser qu'une seule image de machine virtuelle pouvant être déployée sur un parc de machines hétérogènes d'un point de vue tant matériel que de système d'exploitation. Un autre avantage de cette virtualisation est de permettre le contrôle des ressources mises à disposition des machines virtuelles. En effet, il est possible, lors de la création d'une machine virtuelle, de spécifier les ressources qui lui seront allouées, comme par exemple le nombre de processeurs, la quantité de mémoire et d'espace de stockage ;
- un système de contrôle : cette partie fournit des mécanismes permettant la gestion des machines virtuelles. Ces mécanismes permettent dans un premier temps de gérer une machine virtuelle comme une machine physique, en la démarrant, en l'arrêtant, ou en la redémarrant. D'autres fonctions viennent s'ajouter à celles-ci, comme par exemple la mise en pause de la machine virtuelle. Cette fonction permet de stopper la machine virtuelle tout en conservant son état (processus en cours d'exécution, connexions réseau ouvertes...) en vue d'un redémarrage. À la suite de ce redémarrage, la machine se comporte comme si elle n'avait jamais été arrêtée. Des fonctions sont également disponibles afin de pouvoir interagir avec le système virtualisé, comme par exemple l'exécution d'une commande dans la machine virtuelle, ou le dépôt et le rapatriement de fichiers.

Avec un hyperviseur de type 2, les systèmes invités ont l'impression de s'exécuter sur une machine réelle – ils n'ont pas conscience d'être exécutés dans un environnement virtuel. Ceci n'implique donc aucune modification au niveau de ces systèmes, qui s'exécutent normalement.

Cette technique de virtualisation est comparable à un émulateur, et bien souvent elle est confondue avec un tel logiciel. Cependant, avec cette technique les ressources, c'est-à-dire les unités de calcul (processeurs et cœurs de calcul), la mémoire ainsi que l'espace de stockage (ici via l'utilisation d'un ou plusieurs fichiers), sont directement accessibles par la machine virtuelle au travers de pilotes spécifiques. Avec l'utilisation d'un émulateur, toutes les ressources sont simulées, impliquant des pertes conséquentes de performances. Le fait que cette technique de virtualisation permette à la machine virtuelle d'accéder directement aux ressources augmente les performances de celle-ci. Il est à noter ici que bien souvent, les fonctionnalités des processeurs « virtuels » sont en partie « bridées », afin de permettre la portabilité des machines virtuelles. Cette limitation représente un coût qui peut être non négligeable. Ainsi, certains hyperviseurs (comme VmWare par exemple), permettent de limiter les fonctionnalités disponibles sur un

processeur (jeux d'instructions, 32/64 bits...). Ceci permet de conserver l'accès direct aux processeurs, en perdant certaines fonctionnalités, mais en offrant de plus fortes possibilités de portabilité.

L'usage de cette technique permet de mieux isoler les machines virtuelles du système hôte. Ainsi les programmes s'exécutant dans la machine virtuelle peuvent plus difficilement être atteints par une attaque malveillante provenant du système hôte, et vice versa. Un autre avantage de cette technique est que les machines virtuelles, et par extension les applications s'y exécutant, peuvent utiliser des canaux standards de communication, comme le protocole TCP/IP. Ces canaux de communication peuvent être utilisés aussi bien pour que les machines virtuelles échangent entre elles, qu'avec l'hôte ou bien d'autres machines. Ceci est rendu possible par la création d'un tampon d'échange créé par l'hyperviseur, afin de créer des cartes réseaux virtuelles sur une seule carte réelle.

Pour la mise en œuvre de cette technique, il existe de nombreux logiciels, disponibles pour quasiment tous les systèmes d'exploitation. Ces solutions sont : *VirtualPC* et *VirtualServer* (Microsoft), *Parallels Desktop* et *Parallels Server*, *VirtualBox* (Oracle ; gratuit et sous licence libre), et les environnements *VmWare* [5] *Fusion*, *Player* (gratuit), *Server* et *Workstation*.

6.3.4 Hyperviseur de type 1

Dans cette section nous présentons la dernière technique majeure de virtualisation, qui consiste en l'utilisation d'un hyperviseur de type 1. Contrairement à l'hyperviseur de type 2, celui-ci n'est pas un logiciel exécuté par le système hôte, mais il peut être considéré comme une sorte de système d'exploitation.

La figure 6.4 présente principe d'utilisation d'un hyperviseur de type 1.

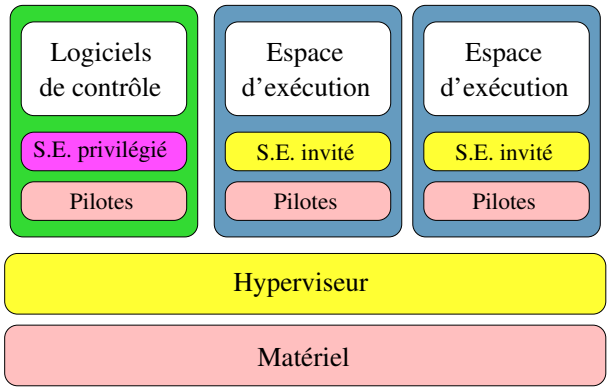

FIGURE 6.4 – Principe de fonctionnement d'un hyperviseur de type 1

Comme on peut le remarquer sur la figure, l'hyperviseur de type 1 est installé directement sur le matériel, comme un système d'exploitation. Cependant, ce n'est pas un système d'exploitation à proprement parler, mais un noyau très léger et optimisé qui a en charge la gestion des accès aux ressources. Tous les systèmes invités passent par l'hyperviseur pour y accéder. En ce sens, les systèmes invités ont conscience qu'ils sont virtualisés, contrairement à l'utilisation d'un hyperviseur de type 2. Ceci implique l'utilisation de systèmes intégrant ce concept, appelé *paravirtualisation*. Pour utiliser ce concept, les systèmes d'exploitation utilisés dans les machines virtuelles doivent être « portés » (ou adaptés) à l'exécution sur un hyperviseur de type 1.

La paravirtualisation est un mécanisme permettant de présenter une interface logicielle similaire à du matériel aux machines virtuelles. Les systèmes d'exploitation de celles-ci doivent en tenir compte, et ne doivent pas faire appel aux ressources directement, mais doivent formuler des requêtes d'accès à

l'hyperviseur. Une fois cet accès autorisé, la machine virtuelle peut accéder de manière transparente aux ressources, mais toujours sous le contrôle de l'hyperviseur.

Comme on peut le remarquer sur la figure 6.4, il existe un système d'exploitation privilégié. Celui-ci est exécuté dès le démarrage de la machine ; il est exécuté par l'hyperviseur lui-même. Ce système permet à l'administrateur de la machine de contrôler les machines virtuelles. Le contrôle se limite ici aux opérations de gestion des machines, et non des interventions au sein des machines elles-mêmes. Les opérations possibles sont par exemple le démarrage, l'arrêt ou le redémarrage. En plus de ces opérations classiques, des fonctions telle que la migration sont disponibles. Cette dernière permet de déplacer l'exécution d'une machine virtuelle sur un autre hôte, moins chargé par exemple. Certains hyperviseurs, comme Xen [19], permettent d'effectuer le transfert d'une machine virtuelle en cours d'exécution, avec un très faible impact (une coupure de quelques secondes dans le pire des cas).

Cette technique de virtualisation est la plus performante qui soit, car elle permet d'utiliser de nombreuses machines virtuelles, qui peuvent accéder de manière presque directe aux ressources. Ceci leur permet d'obtenir des performances quasi similaires à celles qu'elles peuvent atteindre si elles n'étaient pas virtualisées. Cependant, les inconvénients de cette technique ne sont pas négligeables. L'inconvénient majeur est le fait que chaque machine hôte doit être installée et configurée spécifiquement afin de permettre l'utilisation d'un hyperviseur de type 1. De plus, un administrateur doit surveiller et gérer les machines virtuelles qui sont exécutées. Un autre inconvénient majeur est que les systèmes d'exploitation des machines virtuelles doivent être adaptés afin de fonctionner sur ce genre d'hyperviseur. Or à ce jour, peu de systèmes d'exploitation propriétaires sont capables de fonctionner correctement sur une telle architecture, voire de simplement s'exécuter. Cependant, l'obligation d'adapter les systèmes d'exploitation à l'utilisation d'un hyperviseur tend à disparaître. En effet, les fabricants de processeurs, notamment Intel et AMD, proposent des instructions de virtualisation matérielle directement intégrées aux processeurs (Intel-VT et AMD-V respectivement) qui permettent de faire transiter les demandes d'utilisation des ressources par le biais de l'hyperviseur.

Pour la mise en œuvre de cette technique de virtualisation, plusieurs environnements sont proposés. On peut citer par exemple : *Xen* [19] (libre), les environnements VmWare [5] *ESX* et *ESXi*, *Hyper-V Server* (Microsoft), *KVM* (libre), et *Oracle VM* (gratuit mais non libre).

6.4 Plateformes de calcul haute performance utilisant des techniques de virtualisation

Après avoir présenté les principales techniques de virtualisation, nous donnons dans cette section deux exemples de plateformes de calcul hautes performances utilisant ces techniques. Ces plateformes sont *i*Share et Harmony. Chacune de ces plateformes utilise une technique de virtualisation différente.

6.4.1 *i*Share

La première plateforme de calcul que nous présentons est *i*Share [58]. Cette plateforme est dédiée à l'utilisation de machines mises à disposition sur Internet par des utilisateurs, qu'ils soient des particuliers ou des entreprises. Le modèle sur lequel repose *i*Share est basé sur celui qu'utilisent les projets comme Seti@Home, à savoir le volontariat des utilisateurs. Ceux-ci se rassemblent au sein d'une communauté et proposent de partager leurs ressources disponibles, afin de bénéficier d'une plus grande puissance de calcul pour l'exécution d'applications parallèles. La figure 2.4 donnée dans la section 2.1.4 du présent document représente une telle architecture.

Pour le déploiement des ressources, la plateforme fournit tous les outils nécessaires permettant la configuration des machines offrant des ressources, la publication de celles-ci au sein de la plateforme et

leur utilisation. Le point important ici est que lors de la création d'une ressource partagée, le fournisseur de celle-ci peut indiquer quels en sont les attributs. Ceux-ci peuvent représenter la quantité de mémoire qui est partagée, le nombre de processeurs, ou l'espace disque alloués à la plateforme. L'espace disque fourni par les machines permet de stocker les images des machines virtuelles et les applications avec leurs données. La plateforme utilise une structure pair-à-pair (P2P) afin de décentraliser sa gestion des ressources et de ne pas utiliser un serveur central. Chaque utilisateur de la plateforme a accès à un portail lui permettant de sélectionner les machines qu'il souhaite utiliser pour exécuter ses calculs. Il est à noter ici que les utilisateurs fournissant des ressources ont des accès privilégiés à la plateforme par rapport à ceux étant de simples utilisateurs.

La technique de virtualisation utilisée par la plateforme est celle du noyau en espace utilisateur. L'outil qui a été retenu est UML (*User Mode Linux*). Comme nous l'avons indiqué dans la section 6.3.2, cette technique ne permet pas une isolation complète des calculs effectués dans l'environnement virtuel et n'est pas indépendante du matériel de l'hôte. Ainsi, un utilisateur doit prendre garde lors de la sélection des machines de la plateforme à la compatibilité de son application avec ces dernières.

Dans un tel environnement, les ressources sont volatiles et peuvent se déconnecter à n'importe quel moment. La plateforme tient compte de ce paramètre lors de la procédure d'allocation des ressources. En effet, lorsqu'un utilisateur émet une demande de réservation d'un certain nombre de machines ayant des caractéristiques bien définies, un module de gestion des ressources effectue une recherche. Cette recherche est basée tout d'abord sur les caractéristiques voulues par l'utilisateur, mais aussi sur un historique des machines. Pour chaque machine, une trace de sa disponibilité est conservée. Une machine sera plus souvent sélectionnée si celle-ci se déconnecte peu, même si elle est peu puissante, alors qu'une machine puissante ayant une faible disponibilité le sera moins souvent. Le temps d'exécution de l'application est évalué et les machines sont sélectionnées en conséquence. Ceci permet d'éviter statistiquement les pannes durant le calcul. Cependant, si une panne survient durant le calcul, celui-ci doit être de nouveau exécuté. Une fois les machines sélectionnées, la plateforme démarre sur celles-ci les machines virtuelles et y transfert l'application avec ses données.

La plateforme *i*Share permet d'utiliser des applications dites communicantes, grâce à une couche de communication dédiée décrite dans [52]. Cette couche de communication est directement intégrée dans le noyau exécuté en espace utilisateur (ici UML). Le mécanisme mis en place permet de cloisonner les communications entre les machines virtuelles utilisées pour l'exécution d'une application. Lors de leur configuration, durant la phase de déploiement précédemment décrite, chaque machine virtuelle reçoit les adresses de toutes les machines participantes, et n'autorise la réception et l'envoi de données que vers celles-ci sur des ports spécifiques.

Cette plateforme permet de fédérer les ressources des machines de personnes participant volontairement à la construction d'une communauté de calcul. Ses avantages sont la facilité de déploiement de l'architecture (pas d'hyperviseur de type 1) et la sécurisation des liens de communications. Cependant, certains de nos objectifs ne sont pas satisfaits avec l'utilisation de cette plateforme. Le premier d'entre eux est la préservation de la confidentialité des données. En effet, un utilisateur de cette plateforme ne peut pas être sûr de ce qu'il se passe sur les machines de la plateforme, et la sécurité de l'exécution des applications est faible de par l'usage de UML (noyau en espace utilisateur). De même, le second critère non satisfait provient de l'utilisation d'un noyau en espace utilisateur ne permet pas de s'affranchir de la dépendance au matériel de la machine hôte, obligeant ainsi l'utilisateur à gérer lui-même l'hétérogénéité des machines. Enfin, le dernier critère non satisfait repose sur l'absence de mécanismes de tolérance aux pannes adaptés aux applications de calcul scientifique communiquantes. En effet, la politique de la plateforme *i*Share est de redémarrer une tâche depuis le début lorsque la machine l'exécutant devient indisponible. Ceci entraîne la perte de cohérence des calculs et des données au sein des applications avec des communications entre les tâches.

6.4.2 Harmony

La seconde plateforme de calcul que nous présentons est Harmony [51]. L'objectif de cette plateforme est de fédérer les machines de travail disponibles au sein d'une entreprise, afin de bénéficier de leurs ressources inutilisées. Ces ressources sont alors utilisées principalement pour effectuer des traitements sur des bases de données, en utilisant des applications de gestion ou produisant des statistiques par exemple. Les applications visées sont des applications reposant sur le modèle transactionnel.

L'architecture de la plateforme se compose de trois entités : un gestionnaire de requêtes, une couche logique de gestion des ressources, et des démons, ayant le rôle de gestionnaires des ressources physiques. Le gestionnaire de requêtes est la partie en lien avec les utilisateurs. Ceux-ci s'y connectent via une interface de services et peuvent effectuer des demandes d'exécution d'applications. Lorsqu'une requête est effectuée, celle-ci est transmise à un ordonnanceur. Celui-ci va effectuer une demande de réservation de ressources au service gérant la couche logique de gestion des ressources.

Cette couche logique de gestion des ressources utilise une base de données des machines disponibles. Dans celle-ci sont présentes des informations sur chacune des machines : ses caractéristiques (nombre de processeurs et leur fréquence, taille de la mémoire, espace de stockage ...), ses disponibilités, et des statistiques sur ses disponibilités. Chaque ressource mise à disposition de la plateforme peut configurer ses disponibilités. Celles-ci représentent les plages où la plateforme peut l'utiliser et également quelles sont les ressources allouées. Les statistiques sur ces disponibilités viennent ajouter des données représentant l'historique d'utilisation de la machine. Une machine ayant de bonnes statistiques (souvent connectée et possédant des ressources performantes) sera sélectionnée en priorité. Ce modèle statistique permet aussi d'évaluer le temps d'exécution de l'application demandée. Cette couche est aussi responsable du déploiement des machines virtuelles, qui sont gérées par des démons placés sur les machines physiques.

Les démons sont en charge de la gestion des machines virtuelles et de transmettre les données relatives à la charge des machines physiques à la couche logique de gestion des ressources. Lorsque cette dernière indique à un démon de démarrer une machine virtuelle contenant une application spécifique, celui-ci démarre l'image correspondante. La technique de virtualisation qui est utilisée repose sur un hyperviseur de type 2, qui est VMWare Workstation. Au sein du système d'exploitation virtualisé est exécuté un programme permettant de transmettre au démon la charge de la machine virtuelle, qui sera transmise ensuite à la couche logique de gestion des ressources – ces informations lui permettent d'ajuster les estimations des temps d'exécution des applications. Lorsqu'une machine virtuelle devient indisponible (un utilisateur travaille sur la machine physique, ou celle-ci est tombée en panne), la couche logique de gestion des ressources choisit une autre machine pour redéployer le service perdu. Dans le cas où les machines disponibles possèdent de mauvaises statistiques, ou si les statistiques prédisent une faible possibilité de terminaison de l'application, plusieurs instances de l'application sont exécutées.

Cette plateforme est performante lorsque les applications utilisées sont transactionnelles et dont le temps d'exécution n'est pas trop long. De plus, de par l'utilisation d'un hyperviseur de type 2, les calculs et les données préservent leur confidentialité. Les applications ciblées par Harmony sont donc soit transactionnelles, soit des applications de calcul dont les tâches de calcul sont indépendantes. Cependant cette plateforme ne permet pas de satisfaire tous nos critères. Premièrement, sa mise œuvre implique à l'utilisateur de mettre en place des bases de données pour stocker les informations et statistiques relatives aux machines hôtes. Ceci implique la mise en place d'une machine dédiée et de mécanismes de sauvegarde de celle-ci. Deuxièmement, la politique de tolérance aux pannes mise en place, en répliquant les tâches de calcul, impose à l'utilisateur de gérer les doublons de communications pouvant se produire. C'est donc à l'utilisateur d'intégrer une partie de la politique de tolérance aux pannes au sein du code de son application. Cette politique de tolérance aux pannes ne permet pas d'assurer la terminaison d'une application de calcul avec des tâches communiquant entre elles, notamment dans le cas de pannes multiples, car aucun mécanisme n'est prévu à cet effet.

6.5 Conclusion

Dans ce chapitre nous avons présenté l'état de l'art concernant la virtualisation et les plateformes de calcul utilisant des machines virtuelles.

Dans un premier temps nous avons décrit les motivations menant à l'utilisation de solutions de virtualisation pour la création de plateformes de calcul. Ces motivations découlent du fait que dans les institutions actuelles, qu'elles soient des entreprises ou des établissements publics, les parcs de machines de travail sont bien souvent sous-exploités. En effet, les machines sont de plus en plus puissantes et bien souvent leur potentiel n'est pas utilisé, car les besoins des utilisateurs ne changent presque pas. Nous pensons qu'il serait dommage de ne pas profiter de cette puissance de calcul dormante afin d'exécuter des applications scientifiques.

Les objectifs de notre approche consistent à fournir à des utilisateurs non informaticiens une plateforme de calcul simple à mettre en place, à utiliser et tolérante aux pannes. Nous avons montré en quoi l'utilisation de techniques de virtualisation permettraient la mise en œuvre de notre plateforme.

Nous avons ensuite présenté les principales techniques de virtualisation existantes. Nous avons décrit ces techniques en partant de la moins intrusive (celle nécessitant le moins d'intervention de la part de l'administrateur des machines) à celle nécessitant plus de modifications et de configuration. Chaque technique permet une isolation plus ou moins importante des applications de calcul. Les deux seules techniques permettant une isolation forte et sécurisée des exécutions des applications, sont basées sur l'utilisation d'un hyperviseur. L'utilisation d'un hyperviseur de type 2 est simple dans le sens où celui-ci s'installe comme un logiciel classique, et ne nécessite donc pas de configuration particulière de la part de l'administrateur. Cependant, comme il propose une couche de virtualisation du matériel, cette solution tend à perdre en performance de calcul et de liaison réseau. L'avantage de cette technique est de rendre les machines virtuelles portables, c'est-à-dire qu'elles peuvent être exécutées de la même manière sur des machines hôtes d'architectures différentes.

Nous avons également présenté deux plateformes de calcul utilisant des machines virtuelles. Chacune de ces plateformes utilise une technique de virtualisation différente. La première, *i*Share, utilise la technique mettant en place l'exécution d'un noyau en espace utilisateur, en utilisant UML (*User Mode Linux*). Comme nous l'avons vu, cette technique n'offre pas une isolation totale des applications s'exécutant dans l'environnement virtuel. Les objectifs de cette plateforme sont de fédérer les machines mises à disposition par des personnes sur Internet. Des mécanismes permettent d'utiliser des applications communicantes en créant des tunnels identifiés empêchant ainsi l'insertion de données frauduleuses durant le calcul. Le modèle de tolérance aux pannes repose sur une étude statistique prenant en compte la fréquence et la durée des pannes des machines constituant la plateforme. Il n'y a aucun mécanisme de prévu pour pallier la perte d'une machine au cours de l'exécution d'une application de calcul. La seconde plateforme, Harmony, permet de fédérer les machines au sein d'une institution, comme une entreprise par exemple, afin d'utiliser les ressources non utilisées. Cette plateforme utilise la technique de virtualisation reposant sur un hyperviseur de type 2, en utilisant VMWare Workstation. Les applications visées par cette plateforme sont du type transactionnel, c'est-à-dire qui effectuent des traitements sur des données présentes dans des bases de données. L'architecture et la conception de cette plateforme ne sont pas adaptées à l'exécution d'applications de calcul utilisant des communications régulières, ou alors en obligeant l'utilisateur à modifier son application pour qu'elle résiste aux pannes. En effet, la politique de tolérance aux pannes mise en place dans la plateforme consiste en la duplication des tâches de calcul, et ce uniquement dans le cas où le temps d'exécution estimé de l'application se situe dans un intervalle dans lequel il est statistiquement possible que des machines tombent en panne au cours de l'exécution. Ces deux plateformes, bien que performantes dans leur domaine d'application, ne satisfont pas les objectifs que nous nous sommes fixés.

Chapitre 7

Plateforme HpcVm

Dans ce chapitre nous présentons notre plateforme HpcVm, pour « High Performance Computing with Virtual Machines », pour le calcul haute performance sur des architectures de type « enterprise desktop grid ». Il est à noter que cette plateforme n'est pas dédiée à l'exécution d'applications itératives asynchrones, comme l'est JaceP2P-V2. Elle permet d'exécuter tous types de calculs parallèles. Dans un premier temps nous présentons les choix techniques que nous avons effectués pour la mise en œuvre de cette plateforme. Ensuite, nous décrivons son architecture et son fonctionnement. Enfin, la dernière section présente les expérimentations que nous avons menées afin d'évaluer ses performances.

7.1 Choix techniques

Dans cette section nous présentons les choix techniques que nous avons effectués afin de respecter nos objectifs, décrits dans la section 6.1.

7.1.1 Langages de programmation

Concernant le langage utilisé pour la réalisation de la plateforme de calcul, notre choix s'est porté vers Java. L'utilisation de ce dernier rend la plateforme portable, du moins sur tous les systèmes d'exploitation pris en charge par le langage Java. Ainsi, le déploiement de la plateforme ne nécessite pas d'installer différentes versions, prévues pour des systèmes d'exploitation et des configurations matérielles spécifiques, mais seulement le même logiciel. L'autre avantage lié à l'utilisation de ce langage est le fait que pour l'exécution des applications, celles-ci se trouvent dans une machine virtuelle, la JVM (*Java Virtual Machine*). Ceci permet d'isoler l'exécution de l'application Java du reste de l'environnement. Ainsi, comme il est en plus possible de « brouiller » l'exécutable Java (on parle ici d'obfuscation du code), il est très difficile pour un utilisateur lambda d'interagir avec la plateforme s'il n'y a pas d'accès, pour la modifier ou pour perturber les calculs s'y exécutant.

Pour les applications de calcul des utilisateurs, le choix leur est laissé libre. En effet, ils peuvent non seulement choisir le langage de programmation qu'ils souhaitent, mais ils peuvent également utiliser le système d'exploitation qu'ils préfèrent. Pour les besoins de gestion et de fonctionnement de la plateforme, il est cependant nécessaire d'ajouter quelques fonctions. Ces fonctions servent notamment à configurer le réseau de la machine virtuelle (un programme est donc nécessaire au niveau du système

d'exploitation), et une fonction doit être intégrée à l'application de calcul de l'utilisateur. Celle-ci prend en paramètre l'identifiant de la tâche au sein du calcul, une commande (indiquant soit le démarrage du calcul, sa terminaison, ou un appel à la fonction de sauvegarde de la plateforme), et l'adresse IP ainsi que le port de communication avec la machine hôte (la communication ne se fait qu'avec l'élément de gestion de la plateforme). Ces deux dernières informations sont transmises par la plateforme lors du démarrage de la machine virtuelle. Ainsi, le code de l'application de l'utilisateur doit subir une légère modification, qui est l'ajout d'au moins deux appels de fonction (une pour indiquer le démarrage de l'application et l'autre pour indiquer sa terminaison), et un troisième pour une éventuelle demande de sauvegarde.

7.1.2 Technique de virtualisation

Dans cette section nous présentons la technique de virtualisation que nous avons retenue ainsi que le logiciel de virtualisation que nous avons choisi d'utiliser.

Pour les machines virtuelles, nous avions le choix entre les hyperviseurs de type 1 et ceux de type 2. Nous avons restreint le choix entre ces deux techniques car ce sont les deux seules permettant une migration des machines virtuelles et proposant le meilleur isolement par rapport aux machines hôtes. Parmi ces deux techniques de virtualisation, notre choix s'est porté sur celle utilisant un hyperviseur de type 2, décrit dans la section 6.3.3. Ce choix nous permet d'atteindre nos objectifs de simplicité d'installation et de déploiement de la plateforme. En effet, avec une telle technique de virtualisation, il n'y a pas besoin de réinstaller les machines hôtes afin de mettre en place un hyperviseur de type 1, puis d'installer et de configurer les différents systèmes d'exploitation et de gestion des machines virtuelles. Avec un hyperviseur de type 2, il suffit d'installer le logiciel de gestion, comme une application classique – la mise en œuvre est donc beaucoup plus simple et les administrateurs des parcs de machines sont souvent plus enclin à installer un logiciel sur tout le parc plutôt que de réinstaller et reconfigurer toutes les machines du parc. Il est donc plus aisé pour un utilisateur de demander l'installation d'un logiciel plutôt que la réinstallation des machines.

Concernant le choix de l'hyperviseur de type 2, nous avons essayé plusieurs logiciels pouvant s'exécuter sur plusieurs systèmes d'exploitation, ceci afin de ne pas restreindre le nombre de machines pouvant potentiellement rejoindre la plateforme. Les logiciels que nous avons retenus sont : *VMWare Player*, *VMWare Workstation*, et *Oracle VirtualBox*. Nous avons effectué des campagnes de test des fonctionnalités de ces environnements, afin de déterminer les possibilités offertes par chacun d'eux concernant les points suivants :

- la création des machines virtuelles et leur déploiement ;
- les communications possibles entre les machines et leurs performances ;
- leur comportement lors de la mise en pause d'une machine virtuelle et du redémarrage de celle-ci sur une machine hôte différente.

Nous avons tout d'abord éliminé le logiciel *Oracle VirtualBox* avec le dernier test, concernant le redémarrage d'une machine virtuelle, mise en pause au cours de son exécution, sur une autre machine hôte. En effet, pour ce test seules les deux solutions *VMWare* permettent de maintenir les connexions réseau ouvertes, et les calculs en cours d'exécution continuent sans rien remarquer – les connexions et les adresses IP des machines virtuelles sont préservées. Il est à noter que ce test a pu être effectué car la bibliothèque MPI que nous utilisons, OpenMPI, suspend les communications entre les tâches durant un certain laps de temps lorsque l'une d'elle est temporairement indisponible, permettant d'attendre le retour de la tâche suspendue. En plus des résultats de ce test, une étude menée dans [27] confirme notre choix en montrant que les solutions *VMWare* offrent les meilleures performances, aussi bien au niveau de la puissance de calcul disponible pour les applications de la machine virtuelle, que pour les performances des communications. Elles obtiennent également le plus faible impact pour les autres utilisateurs de

la machine hôte. Finalement, notre choix s'est porté sur le logiciel *VMWare Player*, qui est gratuit contrairement à *VMWare Workstation* – ceci permet de ne pas induire de coût supplémentaire pour l'utilisateur ou l'institution hébergeant la plateforme.

7.1.3 Technique de sauvegarde

Comme nous souhaitons fournir aux utilisateurs un environnement tolérant aux pannes, nous avons mis en place un mécanisme permettant de supporter les pannes des machines de calcul. Ce mécanisme est basé sur le principe de la sauvegarde/restauration, dont les détails ont été donnés dans la section 1.4.4 du chapitre 1.

Le choix du point de sauvegarde est laissé à l'utilisateur. Par exemple, un emplacement stratégique peut être la fin d'une itération, permettant de préserver la cohérence des données et des phases de calcul lors du redémarrage de l'application, soit après une phase de sauvegarde soit après une panne. Le mécanisme de sauvegarde mis en œuvre au sein de la plateforme est détaillé dans la section 7.2.2.

Pour les phases de transfert des sauvegardes, nous avons mené une étude comparative sur les temps d'exécution de différentes techniques. Nous avons utilisé une machine virtuelle témoin, dont la taille est de 2,1Go. Les réseaux utilisés sont : 1Gbit/s au sein d'une même salle, et deux salles sont reliées entre elles avec un réseau à 100Mbit/s. Nous avons comparé les temps de compression, de décompression, et de transfert des méthodes suivantes : création de l'archive avec *tar* puis compression avec *gzip*, création d'une archive compressée avec *tar* (avec l'option -z), création d'une archive avec *zip*, et enfin la copie et le transfert de la machine virtuelle sans archivage ni compression. Les mesures ont été effectuées entre deux salles, reliées par des liens à 100Mbits/s. Dans le tableau 7.1, chaque méthode est indiquée respectivement par les termes suivants : tar-gz, tgz, zip, et copie. Ce tableau présente les différents temps de chaque méthode, avec les temps de compression, de décompression et de transfert. Il indique aussi pour chaque méthode le temps total d'une opération de sauvegarde et celui d'une opération de récupération d'une sauvegarde.

Phase \ Méthode	tar-gz	tgz	zip	copie
Compression	150 s	105 s	105 s	–
Décompression	95 s	45 s	45 s	–
Copie	–	–	–	40 s
Transfert	15 s	15 s	15 s	31 s
Total sauvegarde	165 s	120 s	120 s	71 s
Total récupération	110 s	60 s	60 s	71 s

TABLE 7.1 – Temps de d'exécution des solutions d'archivage, de compression et décompression, et de transfert des sauvegardes d'une machine virtuelle sur un réseau à 100MBits/s

Comme on peut le remarquer, pour la phase de sauvegarde, c'est la méthode de copie simple qui prend le moins de temps, avec 71 secondes. Ensuite viennent les méthodes tgz et zip, qui effectuent la création d'une archive compressée en 120 secondes. Concernant la phase de récupération, ce sont cette fois les méthodes tgz et zip qui offrent le temps le plus court, avec 60 secondes. Ensuite vient la méthode de copie, avec 71 secondes. La méthode tar-gz (création d'une archive puis compression de celle-ci) offre pour chaque phase le moins bon temps, avec 163 secondes pour la phase de sauvegarde et 110 secondes pour la phase de récupération. Il est à noter ici que les trois méthodes utilisant une compression de la

machine virtuelle permettent d'obtenir une réduction de sa taille à 938Mo. Nous avons volontairement mis de côté la solution utilisant la méthode de copie, car c'est celle qui présente le plus de risques de perturber les calculs, notamment les communications entre les tâches, lors de la phase de transfert. Ce choix a été fait pour le cas où l'architecture sur laquelle la plateforme serait déployée ne bénéficie pas de liens réseau performants et que la taille des machines virtuelles soit plus conséquente. Cependant, des expériences complémentaires permettraient de quantifier cet impact, notamment avec des applications échangeant de grandes quantités de données. Enfin, nous avons choisi, de manière arbitraire, la méthode tgz (bien que la méthode zip présente les mêmes performances).

7.2 Architecture et fonctionnement de HpcVm

Dans cette section nous présentons plus en détails notre plateforme HpcVm. Il est à noter que ceci est la première version de la plateforme, qui bien que fonctionnelle, n'est pour l'instant qu'une démonstration de faisabilité du concept décrit dans la section 6.1. Dans un premier temps nous décrivons son architecture, qui pour l'instant est centralisée, et ensuite nous présentons son fonctionnement, en donnant ses principales fonctionnalités.

7.2.1 Architecture

HpcVm est basée sur le modèle classique client-serveur/démons. Ce modèle a l'avantage de pouvoir être étendu en ajoutant des mécanismes pour la tolérance aux pannes, comme par exemple pour la plateforme JaceP2P-V2, décrite dans la section 2.2.4. La figure 7.1 présente le schéma de son architecture.

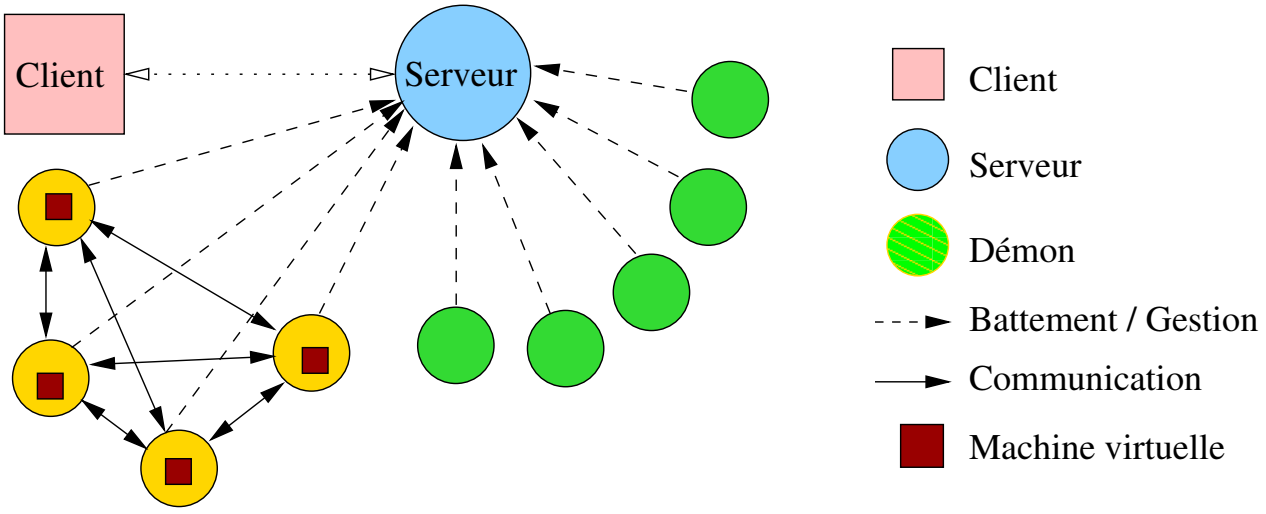

FIGURE 7.1 – Architecture de la plateforme HpcVm

Comme on peut le remarquer sur la figure, la plateforme HpcVm est découpée en trois entités principales : le client, le serveur et les démons. Le rôle de chacune de ces entités est le suivant :

- le serveur : il est le point central de la plateforme. C'est sur lui que se connectent les démons, qui sont exécutés sur les machines hôtes. Le serveur maintient une liste de ces démons et les surveille, via la réception régulière de messages de « battement de cœur ». C'est également le serveur qui est en charge de générer la configuration réseau des machines virtuelles qui seront exécutées et contrôlées par chaque démon. Il assure le bon déroulement des applications et il sert

d'interface au client. Lors de son exécution, il prend en paramètre des informations lui permettant de configurer les machines virtuelles, comme la classe d'adresses IP qui sera utilisée, mais il sert également de serveur de stockage pour les images des machines virtuelles ;

- le client : il est exécutée par l'utilisateur voulant réserver des machines dans le but d'exécuter une application de calcul. Le client intervient en plusieurs étapes pour cette réservation, ainsi que pour le lancement des machines virtuelles. Si le nombre de machines hôtes est suffisant, le serveur renvoie une liste d'adresses IP, correspondant à celles des machines virtuelles à la disposition de l'utilisateur. Ce dernier peut alors se connecter à l'une d'entre elles et lancer l'exécution de son application ;

- les démons : ils sont exécutés sur les machines hôtes mises à la disposition de la plateforme, afin d'y lancer des machines virtuelles. Les démons se connectent au serveur en lui donnant des informations sur la machine hôte, comme par exemple son adresse IP et ses caractéristiques (nombre de processeurs et leur fréquence, quantité de mémoire disponible, espace de stockage ...). Une fois connectés au serveur, les démons lui envoient régulièrement un message de battement de cœur pour lui indiquer qu'ils sont toujours « en vie ». Les démons sont en charge du contrôle des machines virtuelles, en ayant la possibilité d'effectuer les opérations suivantes : démarrer, arrêter, mettre en pause, et redémarrer. Ils sont aussi en charge d'effectuer les sauvegardes des machines virtuelles, lorsque la demande de sauvegarde effectuée par l'application a été autorisée.

Comme on peut le remarquer dans cette description, le serveur est un point central de l'architecture. Nous considérons que cette entité est exécutée sur une machine fiable, avec des liaisons réseau performantes, et mettant à disposition un espace de stockage conséquent dédié à la plateforme. Cet espace de stockage doit être suffisamment grand pour accueillir l'équivalent d'au moins 4 ou 5 images, dont la taille peut varier de 1Go à 4Go (l'espace réservé doit donc être d'au moins 20Go). Les limitations imposées par la supposition forte d'une machine fiable pour le serveur seront amenées à être réduites dans les prochaines versions de la plateforme. Une stratégie de décentralisation, telle que celle mise en œuvre dans la plateforme JaceP2P-V2, pourra être envisagée. Cette décentralisation reposera sur le délestage de certaines opérations effectuées par le serveur, comme la surveillance des machines au profit d'un lanceur et de l'utilisation d'un protocole de bavardage exécuté sur les démons, et de sa réplication afin d'équilibrer la charge et d'assurer la tolérance aux pannes.

7.2.2 Fonctionnement

Dans cette section nous présentons le fonctionnement de la plateforme. Nous donnons par ordre chronologique les différentes étapes, aussi bien d'installation et de configuration, que les étapes de réservation et de déploiement des machines virtuelles, ainsi que l'exécution d'une application de calcul. Nous présentons également les mécanismes de sauvegarde et de tolérance aux pannes qui sont mis en œuvre.

Installation et configuration

La figure 7.2 illustre l'installation de ses composants. Ce déploiement est identique pour le serveur et les démons.

Concernant tout d'abord le serveur, la machine utilisée doit être fiable, et l'utilisateur doit prévoir un espace de stockage dédié à l'utilisation de la plateforme. Il est à noter ici que cet espace ne doit pas être partagé par toutes les machines hôtes comme il pourrait l'être par exemple avec NFS, et doit avoir

une taille raisonnable – environ 20Go. C'est dans cet espace que seront stockées les images initiales des machines virtuelles. L'autre point requis est une installation de l'environnement Java.

Pour l'installation du serveur, il suffit de récupérer l'archive de la plateforme, nommée *HpcVm.jar*. Cette archive contient le programme de la plateforme sous une forme binaire. Pour exécuter le serveur, il suffit de le lancer en lui donnant en paramètre son port d'écoute sur lequel pourront se connecter les démons et les clients, et éventuellement un paramètre indiquant un temps minimum en minutes pour l'espacement des sauvegardes. En effet, les applications peuvent avancer suffisamment vite pour demander de manière trop rapprochée l'exécution des mécanismes de sauvegarde. Ce paramètre permet de limiter l'appel à ces mécanismes. Il est à noter ici que la sauvegarde des machines virtuelles prend un temps non négligeable et l'utilisateur doit faire attention à ce que les sauvegardes ne soient pas trop rapprochées dans le temps, afin d'éviter une perte de performances non négligeable. Une fois ceci fait, le serveur est en attente de connexion des démons.

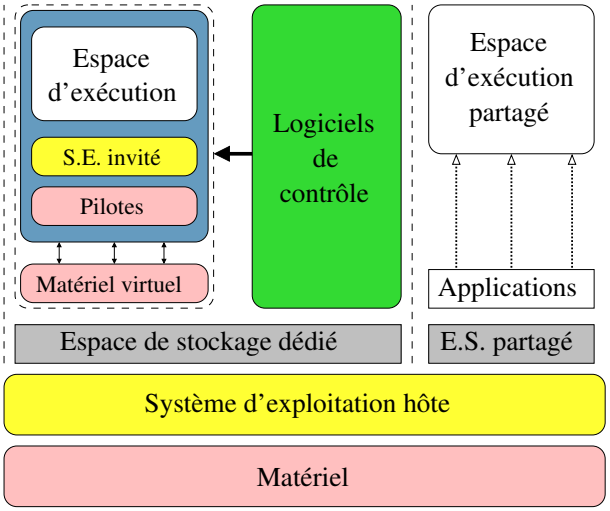

FIGURE 7.2 – Installation des composants de la plateforme HpcVm

Concernant les démons, l'installation requise est identique à celle du serveur, pour rappel un environnement Java installé et configuré, et un espace de stockage dédié. Ici aussi cet espace de stockage ne doit pas être commun à toutes les machines. Ceci provient du fait qu'une seule image de la machine virtuelle va être déployée sur toutes les machines hôtes et si celle-ci est partagée, des conflits seront détectés et les machines virtuelles ne pourront pas démarrer. De même il faut prévoir une place non négligeable car cet espace accueillera l'image initiale de la machine virtuelle, son image de sauvegarde et au moins une image de sauvegarde d'une de ses voisines de calcul. Ici aussi il suffit de récupérer l'archive de la plateforme et de l'exécuter. Les paramètres à indiquer lors de l'exécution du démon sont l'adresse IP du serveur et le numéro de son port d'écoute.

Concernant cette fois la machine virtuelle, le choix se porte pour le moment sur un système d'exploitation GNU/Linux, comme Debian ou Ubuntu par exemple. Ce choix volontaire réside dans le fait que ces systèmes d'exploitation sont libres et surtout gratuits, n'induisant pas un coût financier supplémentaire pour l'utilisateur. L'utilisateur doit configurer une machine virtuelle en installant un système d'exploitation et en définissant ses caractéristiques. La taille de la mémoire allouée doit être la plus petite possible, mais doit permettre de charger toutes les données nécessaires aux calculs. Ceci permet d'augmenter les performances aussi bien pendant l'exécution des calculs (un espace mémoire trop grand peut empêcher le bon fonctionnement de l'hôte et le ralentir, et donc ralentir les calculs), que lors de la phase de sauvegarde (moins de données inutiles devront être sauvegardées et transférées). Il en va

de même pour l'espace de stockage alloué à la machine virtuelle, il doit être le plus petit possible (pour les mêmes raisons que la mémoire). Une fois la machine virtuelle créée et configurée, l'utilisateur doit créer une archive de celle-ci. Il doit ensuite copier cette archive sur le serveur, dans le répertoire dédié à la plateforme. Il peut dès lors également choisir de copier cette archive sur les machines hôtes (afin de réduire le temps de déploiement des machines virtuelles), mais ceci n'est pas obligatoire.

Déploiement des machines virtuelles

Une fois la plateforme et au moins une machine virtuelle configurées, l'utilisateur peut déployer la plateforme ainsi que la machine virtuelle sur toutes les machines hôtes disponibles. La figure 7.3 illustre le déploiement des machines virtuelles au sein de la plateforme.

L'utilisateur doit préparer la plateforme pour l'exécution de son application. Cette préparation se déroule en cinq étapes :

1. l'utilisateur fait appel à une fonction du client qui demande au serveur de déployer l'archive de la machine virtuelle sur toutes les machines hôtes. Il indique le nom de la machine virtuelle, l'emplacement de son archive, et donne un nom pour le déploiement (le nom de l'application par exemple). Ce nom sert à identifier la machine virtuelle au sein de la plateforme ;
2. le serveur va alors contacter chaque démon et lui demander de vérifier s'il possède déjà l'archive de la machine virtuelle. Dans le cas négatif, le serveur lui envoie l'archive. Il est à noter ici que comme cette opération est réalisée en parallèle sur toutes les machines hôtes connectées à la plateforme, nous avons mis en place une sécurité bloquant le serveur lors de l'envoi à un trop grand nombre de machines. Une fois cette limite atteinte, le serveur attend que les machines en cours de déploiement aient reçu l'archive avant de continuer le processus avec les machines restantes. Au cours de ce déploiement, les machines hôtes reçoivent éventuellement l'archive de la machine virtuelle (si elles ne l'ont pas), et la décompressent pour en permettre l'utilisation par VMWare Player. Lors de ce déploiement, chaque démon reçoit l'adresse IP de la machine virtuelle dont il a la gestion. Cette adresse est attribuée par le serveur ;

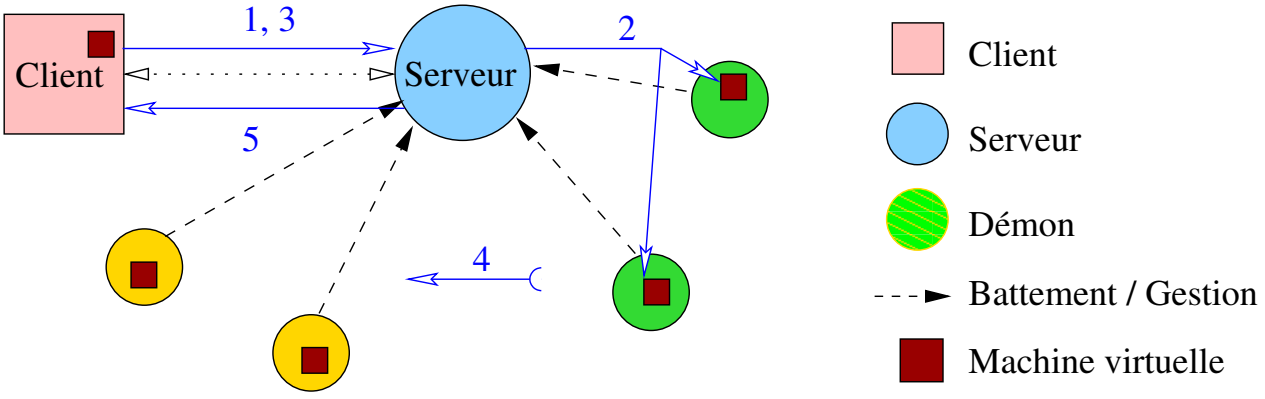

FIGURE 7.3 – Déploiement des machines virtuelles

3. une fois l'archive de la machine virtuelle déployée sur toutes les machines hôtes, l'utilisateur invoque le client en indiquant combien de machines il souhaite réserver pour exécuter son application. Le client se connecte au serveur et exécute la réservation des machines ;
4. le serveur sélectionne alors le nombre de machines voulues et demande aux démons de lancer leur machine virtuelle. Une fois celle-ci démarrée, le démon lui transmet l'adresse IP qui lui a été attribuée pour qu'elle configure son interface réseau. Cette opération est réalisée à l'aide

d'un script que nous avons inséré dans le mécanisme de démarrage du système d'exploitation. Le démon lui transmet aussi son adresse IP ainsi que le numéro de port sur lequel il écoute. Ce port est réservé aux communications entre le démon et la machine virtuelle. Si une machine virtuelle ne démarre pas, le serveur, informé par le démon, sélectionne un autre démon ;

5. une fois que le nombre voulu de machines virtuelles ont démarré, la liste de celles-ci est renvoyée au client qui l'affiche à l'utilisateur.

Exécution d'une application

Une fois que les machines virtuelles sont démarrées et opérationnelles, l'utilisateur reçoit la liste de leur adresse IP. Il lui suffit alors de se connecter sur l'une d'elle, en utilisant par exemple une connexion SSH. L'utilisateur se connecte à un compte qu'il a préalablement créé au moment de la création de la machine virtuelle. Une fois connecté, il peut exécuter son application en lui indiquant les adresses IP des machines virtuelles. L'application de l'utilisateur doit être légèrement modifiée afin de pouvoir profiter des mécanismes de tolérance aux pannes de la plateforme. Trois appels à une fonction de communication avec le démon peuvent être réalisés :

- le premier appel indique le démarrage de l'application, qui permet au serveur de prendre en compte le moment où l'application démarre ;
- le second appel est placé à l'endroit choisi par l'utilisateur pour effectuer une sauvegarde du calcul. Cet emplacement doit être mis de préférence entre deux synchronisations, afin de ne pas introduire d'incohérence lors du redémarrage des machines virtuelles ;
- enfin, le troisième appel indique la fin de l'application. Lorsque l'exécution d'une application prend fin, le serveur demande aux démons de sauvegarder l'état des machines virtuelles et de les arrêter.

Mécanisme de sauvegarde

Le mécanisme de sauvegarde utilisé au sein de la plateforme, illustré par la figure 7.4, se déroule en trois phases :

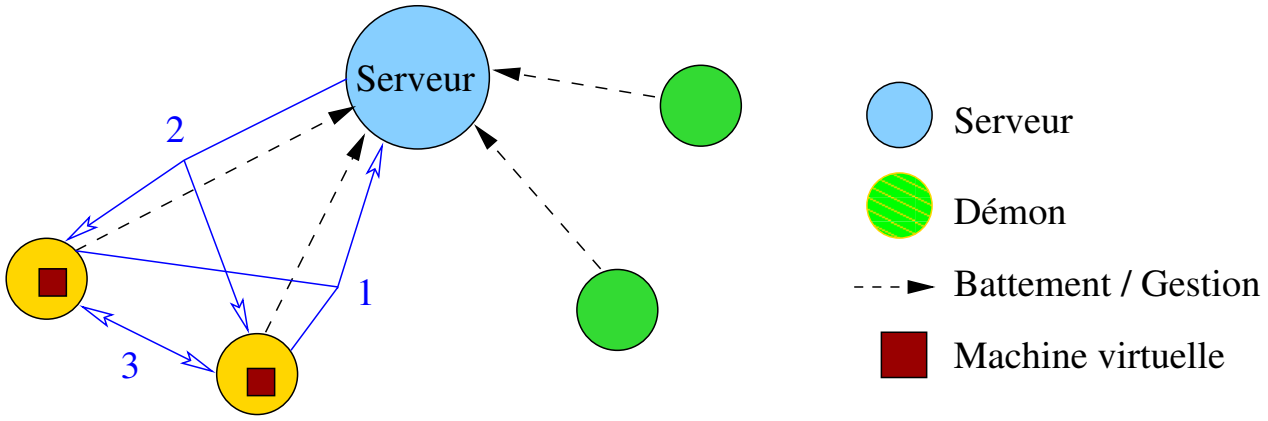

FIGURE 7.4 – Mécanisme de sauvegarde des machines virtuelles

1. les tâches se synchronisent et effectuent une demande de sauvegarde auprès de leur démon qui la transmet au serveur ;
2. le serveur vérifie si les opérations de sauvegarde peuvent être déclenchées. Lors du lancement de l'application, une durée minimale entre les sauvegardes a été renseignée. Si le temps entre

la dernière sauvegarde, ou le démarrage de l'application, est supérieur ou égal à cette durée, le serveur décide d'enclencher la procédure de sauvegarde en envoyant une réponse positive au démon. Sinon, il lui signifie son refus ;

3. lorsque le démon reçoit la réponse du serveur, deux solutions sont envisagées :
   - la réponse est favorable : le démon exécute alors la procédure de sauvegarde. Cette procédure consiste en la mise en pause de la machine virtuelle (la tâche est en attente, donc dans un état cohérent), puis le démon crée une archive de la machine virtuelle. Une fois l'archive créée, la machine virtuelle est redémarrée et le démon envoie un message à l'application lui indiquant qu'elle peut poursuivre son calcul. Ensuite chaque démon envoie une copie de sa sauvegarde à l'une des machines hôtes de la plateforme participant à l'exécution de l'application. Il est à noter que la phase de transfert s'effectue en parallèle des calculs. Une fois toutes les copies envoyées, chaque démon informe le serveur du succès de l'opération et celui-ci déclenche la phase de marquage des archives (sauvegarde globale). Ceci permet de s'assurer que les sauvegardes sont cohérentes ;
   - la réponse est négative : le démon envoie un message à la tâche lui signifiant qu'elle peut poursuivre son calcul.

Le fait d'utiliser un mécanisme de marquage permet d'assurer une garantie sur la conformité de la sauvegarde venant d'être effectuée. Ainsi, lors d'une panne, toutes les tâches peuvent être redémarrées au même niveau de calcul dans un état cohérent. Il est à noter ici que lors des opérations de sauvegarde, dès que les tâches effectuent leur demande de sauvegarde, et jusqu'à ce qu'elles reçoivent un message du démon, elles restent en attente. Ainsi la phase de calcul est bloquée et aucun message ne circule.

Tolérance aux pannes

Lorsqu'une panne est détectée par le serveur durant l'exécution d'une application, celui-ci déclenche la procédure de tolérance aux pannes. Cette procédure, illustrée par la figure 7.5, se déroule de la manière suivante :

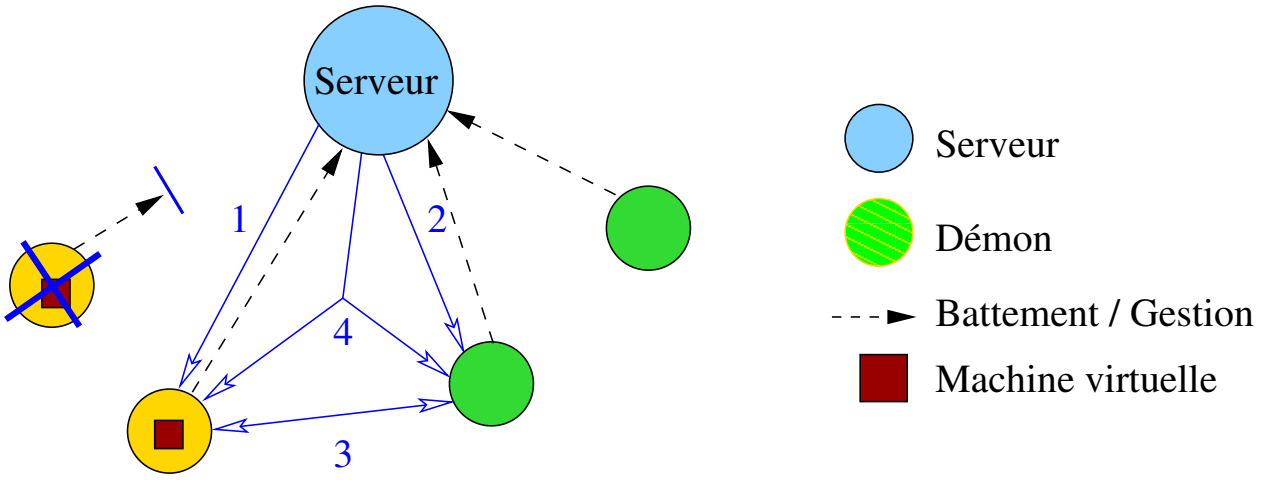

FIGURE 7.5 – Mécanisme de tolérance aux pannes de HpcVm

1. tous les démons sont informés de la panne et arrêtent les machines virtuelles ;
2. le serveur choisit une machine de remplacement et indique au démon s'y exécutant sur quelle machine hôte il peut récupérer la dernière sauvegarde de la tâche exécutée par la machine venant de tomber en panne, en lui fournissant une adresse IP. Le serveur lui envoie également l'adresse

IP de la machine virtuelle venant de tomber en panne, afin que celle-ci puisse être correctement configurée ;

3. le démon exécuté sur la machine hôte remplaçante récupère la sauvegarde en question ;
4. tous les démons, y compris celui s'exécutant sur la machine hôte remplaçante, déploient la dernière sauvegarde en leur possession et démarrent leur machine virtuelle. Le démon en charge de la machine remplaçante configure la machine virtuelle en lui indiquant sa configuration réseau. Ensuite les démons envoient un message aux tâches leur signifiant qu'elles peuvent poursuivre leur calcul.

Ce mécanisme permet de toujours assurer la cohérence des données et des phases de calcul. Ainsi la plateforme permet de supporter la perte de machines hôtes, dans la mesure où il reste suffisamment de machines disponibles pour remplacer celles défaillantes.

7.3 Expérimentations

Dans cette section nous présentons les expérimentations que nous avons menées afin d'évaluer les performances de la plateforme. Le but ici est de montrer que les concepts mis en place dans la plateforme HpcVm sont viables et peuvent être utilisés dans des contextes réalistes. Nous présentons dans un premier temps les conditions de nos expérimentations, en décrivant l'architecture utilisée ainsi que les applications qui ont été exécutées. Dans un second temps nous présentons les résultats de nos expérimentations.

7.3.1 Architecture

Cette section présente l'architecture d'exécution que nous avons utilisé afin de réaliser nos expérimentations. Puis nous décrivons la machine virtuelle que nous avons configurée.

Architecture d'exécution

L'architecture cible est du type « enterprise desktop grid », c'est-à-dire que nous utilisons des machines de travail au sein d'une institution et non une architecture dédiée à l'exécution d'applications de calcul scientifique.

Nous avons utilisé les machines destinées à l'usage des étudiants du département Informatique de l'IUT de Belfort-Montbéliard. Nous avons déployé la plateforme HpcVm sur deux salles de travaux pratiques, comportant chacune 15 machines. Les machines de ces salles sont identiques, et leurs caractéristiques sont les suivantes : processeur AMD X2 comportant 2 cœurs de calcul cadencés à 3Ghz, avec 4Go de mémoire et un espace disque de 250Go. Comme on peut le remarquer, les caractéristiques de ces machines sont surdimensionnées pour des machines servant aux travaux pratiques d'étudiants de DUT (ceci est relatif dans le sens que les applications telles que les navigateurs web ou les IDE actuels requièrent beaucoup de ressources). Elles sont l'exemple typique des architectures qui sont l'objet de notre étude.

Concernant la configuration réseau, chaque salle possède un commutateur réseau de 1Gbit/s reliant toutes les machines entre elles. Les deux salles sont quant à elles reliées ensemble par un commutateur de 100Mbit/s. Les liaisons entre les machines d'une même salle sont donc très performantes et celles entre les machines des deux salles le sont un peu moins, le débit étant moindre. La configuration spécifique du réseau induit des pertes de performances pour les machines virtuelles. En effet, comme nous le décrirons par la suite, nos machines virtuelles sont configurées sur un réseau logique différent

de celui des machines hôtes que nous utilisons, ceci pour des raison de tolérance aux pannes – il est ainsi possible pour les machines virtuelles de conserver leur adresse IP, même si elles sont déplacées. Or, l'architecture réseau mise en place par les administrateurs tend à cloisonner les réseaux logiques utilisables sur les liens physiques pour des raisons de sécurité. Les commutateurs des salles n'opposent aucune barrière contrairement aux commutateurs « intelligents » qui surveillent le réseau. Ces derniers sont configurés pour effectuer les routages entre les différentes salles, et l'utilisation d'un réseau logique annexe (comme celui que nous avons mis en place) est tolérée, mais comme les routes ne sont pas définies, les performances sont réduites.

Machine virtuelle

Pour nos expérimentations, nous avons mis en place une machine virtuelle contenant nos applications avec leurs données. Comme indiqué dans la section 7.1.2, nous utilisons une machine virtuelle VMWare.

Pour les besoins de nos applications, nous l'avons configurée de la manière suivante :

- machine mono-processeur, afin de laisser disponible un cœur de calcul pour les utilisateurs « normaux » des machines hôtes ;
- mémoire vive de 1Go (représentée par un fichier sur la machine hôte) ;
- espace de stockage de 1Go (représenté par un fichier sur la machine hôte). Cet espace comprend la place occupée par le système d'exploitation et par les applications et leurs données ;
- la carte réseau virtuelle est reliée à la carte physique et partage le lien de communication – les machines virtuelles communiquent directement entre elles, sans passer par l'hôte.

Le système d'exploitation que nous avons utilisé est une distribution Debian Squeeze. La taille occupée par la machine virtuelle sur l'espace de stockage dédié à la plateforme est de 2,1Go – ceci représente les différents fichiers (mémoire vive et espace de stockage de la machine virtuelle).

7.3.2 Applications utilisées

Pour évaluer les performances de HpcVm, nous avons utilisé deux applications, ayant chacune un comportement différent, notamment au niveau de leur schéma de communications. Nous avons utilisé l'application de résolution de l'équation d'advection-diffusion, et une application de recherche phylogénique. Chaque application a été découpée en 16 tâches de calcul, réparties sur 16 machines de l'architecture.

Résolution de l'équation d'advection-diffusion

Cette application est décrite dans la section 5.1.3 du chapitre 5. Nous utilisons ici une version itérative synchrone classique, et non plus une version itérative asynchrone. Nous avons choisi cette version, utilisant la bibliothèque OpenMpi [33] pour les communications, car dans le modèle synchrone les tâches se synchronisent à la fin de chaque itération. Ceci nous permet d'effectuer une sauvegarde de la machine virtuelle en s'assurant de la cohérence des données lors du redémarrage de l'application (soit après une panne, soit à la suite de la procédure de sauvegarde). Les tâches de cette application, comme nous l'avons vu lors des expérimentations d'évaluation des algorithmes de placement, voient leur charge de calcul varier au cours de l'exécution et elles s'échangent à chaque itération une quantité de données non négligeable. La matrice que nous avons utilisée dans cette application est une matrice carrée de 800 par 800. Nous avons traité un problème contenant 2 espèces.

Recherche phylogénique

La seconde application que nous avons utilisée est une application de calcul phylogénique, RAxML-HPC [63], pour *Randomized accelerated maximum likelihood for high performance computing*. Les calculs de phylogénie font partie des grands challenges de la bioinformatique, car leur résolution requière de nombreuses ressources de calcul (aussi bien en terme de puissance que de mémoire).

La phylogénie est l'étude des parentés entre différents êtres vivants, comme l'espèce humaine ou plus finement les gènes de l'ADN, et permet de comprendre leur évolution. On peut étudier la phylogénie d'un groupe d'espèces mais également, à un niveau intraspécifique, la généalogie entre populations ou entre individus. Une phylogénie est généralement représentée par un arbre phylogénétique. Le nombre de nœuds entre les branches, qui représentent autant d'ancêtres communs, indique le degré de parenté entre les taxons. Un taxon est défini comme étant une entité conceptuelle regroupant des organismes vivants possédant certaines caractéristiques communes (par exemple on peut définir un taxon pour l'ensemble des personnes ayant des cheveux noirs). Au niveau de l'arbre phylogénique, plus il y a de nœuds, et donc d'ancêtres entre deux espèces, plus leur parenté est éloignée, c'est-à-dire que leur ancêtre commun est ancien. En phylogénie, deux types d'arbres peuvent être utilisés :

- des arbres élaborés par phénétique. La phénétique repose sur le postulat de base que le degré de ressemblance est corrélé au degré de parenté. Elle suppose donc de quantifier la ressemblance entre les êtres vivants à classer. Ainsi, dans un arbre élaboré par phénétique, la longueur des branches représente la distance génétique entre les taxons ;
- des arbres élaborés par cladastique. La cladistique hiérarchise les caractères comparés, c'est-à-dire que ne sont en fait regroupés dans un même taxon que les êtres vivants qui partagent des caractères homologues. Ainsi, dans un arbre élaboré par cladistique, on place sur les branches les évènements évolutifs (caractères dérivés) ayant eu lieu dans chaque lignée.

La méthode utilisée par l'application que nous utilisons cherche à construire un arbre phénétique en optimisant les critères de vraisemblance, avec une approche probabiliste, en se fondant sur le taux de substitution pour chaque élément de base au cours du temps. Elle estime la vraisemblance de la position et de la longueur des branches de l'arbre. De plus amples informations sur l'application peuvent être trouvées dans [3]. Les données que nous utilisons lors de nos expériences utilisent des arbres contenant 200 individus.

7.3.3 Résultats

Dans cette section nous présentons les résultats des expérimentations que nous avons menées, afin d'observer les performances de notre plateforme et de vérifier qu'elle remplisse les objectifs que nous avons définis.

Pour chaque application, nous avons mis en place le même protocole d'expérimentation. Ce protocole consiste en plusieurs exécutions de chaque application, ceci afin d'obtenir une moyenne des temps d'exécution, dans six contextes différents :

- exécutions se déroulant sur les machines hôtes, afin d'obtenir un temps de référence. Ce contexte est noté « Hôtes » dans les tableaux de résultats ;
- exécutions se déroulant dans les machines virtuelles, en inhibant les mécanismes de gestion des demandes de sauvegarde de la plateforme et les demandes de sauvegarde de la part des tâches de calcul. Ceci permet de quantifier le coût induit par l'utilisation de la plateforme et des machines virtuelles. Ce contexte est noté « HpcVm S » dans les tableaux de résultats ;
- exécutions se déroulant dans les machines virtuelles, en activant les mécanismes de gestion des demandes de sauvegarde de la plateforme et les demandes de sauvegarde de la part des tâches de calcul. Ceci permet de mesurer le coût induit par l'utilisation de la plateforme et le surcoût

engendré par les mécanismes de gestion (demandes de sauvegarde et réponses à ces demandes). Ce contexte est noté « HpcVm G » dans les tableaux de résultats ;
- exécutions se déroulant dans les machines virtuelles, avec les mécanismes de gestion de la plateforme activés et le déclenchement d'une seule et de deux sauvegardes. Ces contextes sont notés respectivement « 1 sauv. » et « 2 sauv. » dans les tableaux de résultats ;
- exécutions se déroulant dans les machines virtuelles, avec les mécanismes de gestion de la plateforme activés, le déclenchement d'une seule sauvegarde et l'introduction d'une panne quelques secondes après la terminaison de la phase de sauvegarde. Celle-ci est considérée comme terminée lorsque toutes les sauvegardes ont été envoyées aux voisins de calcul et que ceux-ci en ont confirmé la réception. Ce contexte est noté « 1 s. / 1 panne » dans les tableaux de résultats.

Les résultats de ces expérimentations sont décrits dans les tableaux 7.2 et 7.3. Ces tableaux donnent respectivement les résultats pour les expérimentations avec l'application de résolution de l'équation d'advection-diffusion, et ceux pour l'application de recherche phylogénique. Les résultats comportent les temps moyens d'exécution (en minutes) et le surcoût induit par l'utilisation de la plateforme et du contexte d'exécution (en minutes). Ainsi, la colonne « Hôtes » donne le temps d'exécution de référence de l'application sur les machines hôtes, la colonne « HpcVm S » montre le surcoût lié à l'utilisation de la plateforme, et « HpcVm G » celui lié à l'utilisation de la plateforme avec les mécanismes de gestion de demandes de sauvegarde activés. Les trois dernières colonnes indiquent le temps d'exécution de l'application et le surcoût, par rapport au contexte « HpcVm G », induit par les contextes d'exécutions avec une et deux sauvegardes, et une sauvegarde avec une panne.

Contexte	Hôtes	HpcVm S	HpcVm G	1 sauv.	2 sauv.	1 s./ 1 panne
Temps	63 min	71 min	81 min	86 min	93 min	90 min
Surcoût	–	+ 8 min	+ 18 min	+ 23 min	+ 30 min	+ 27 min
Coûts pannes/sauvegardes			–	+ 5 min	+ 12 min	+ 9 min

TABLE 7.2 – Résultats des expériementations avec l'application de la résolution de l'équation d'advection-diffusion sur la plateforme HpcVm

Tout d'abord, concernant les expériences avec l'application de la résolution de l'équation d'advection-diffusion, son temps moyen d'exécution de référence (avec les machines hôtes) est de 63 minutes. Le temps moyen d'exécution de l'application avec la plateforme, sans les mécanismes de gestion des demandes de sauvegarde est de 71 minutes, induisant un surcoût de 8 minutes. Ce surcoût représente la virtualisation des machines et des communications. Le temps d'exécution de l'application avec la plateforme et les mécanismes de gestion des demandes de sauvegarde activés est de 81 minutes, induisant un surcoût de 18 minutes. On peut remarquer ici que le surcoût induit par les mécanismes de gestion est de 10 minutes. Le temps d'exécution de l'application avec une sauvegarde est de 86 minutes, soit un surcoût de 23 minutes par rapport au temps de référence sur les machines hôtes. L'impact d'une sauvegarde sur le temps d'exécution de l'application est de 5 minutes. Ce temps inclut la mise en pause de la machine virtuelle, sa sauvegarde et son redémarrage. Il est à noter que comme la phase de transfert des sauvegardes est effectuée après le redémarrage des machines virtuelles, entraînant la reprise des calculs, celle-ci impacte les communications entre les tâches et les mécanismes de gestion des demandes de sauvegarde en occupant une partie de la bande passante. Lorsqu'au cours de l'exécution de l'application deux sauvegardes sont effectuées, le temps d'exécution est de 93 minutes, soit un surcoût de 12 minutes. On peut donc en déduire que le coût induit par la seconde sauvegarde est de 7 minutes, ce qui est très proche de celui induit par la première sauvegarde. La différence peut être expliquée par le fait que nous faisons la moyenne des temps d'exécution pouvant induire une telle différence – on peut ainsi considérer le temps moyen d'une sauvegarde comme représentant un coût de 6 minutes pour

cette application. Pour le dernier contexte d'exécution, dans lequel nous effectuons une sauvegarde suivie d'une panne d'une des machines hôtes, le temps d'exécution de l'application est de 90 minutes, représentant un surcoût de 9 minutes. Ainsi, comme le temps d'une sauvegarde est de 6 minutes, on en déduit que le coût d'une panne pour cette application est de 3 minutes. Le temps induit par une panne représente la détection de la panne et le redéploiement des sauvegardes (extraction des sauvegardes et redémarrage des machines virtuelles). Le fait que le coût d'une panne soit de 3 minutes indique que la politique de tolérance aux pannes que nous avons mis en place est efficace.

Les surcoûts que l'on peut observer ici ne dépendent pas exclusivement de l'utilisation de la plateforme et des machines virtuelles. En effet, comme nous l'avons indiqué dans la présentation de l'architecture utilisée pour la réalisation de nos expérimentations, le réseau qui relie les machines oppose quelques obstacles à l'obtention de bonnes performances de communication. Comme les tâches de cette application communiquent souvent entre elles en s'échangeant des messages volumineux, ceci induit des pertes de performances, et donc un allongement des temps d'exécution.

Contexte	Hôtes	HpcVm S	HpcVm G	1 sauv.	2 sauv.	1 s./ 1 panne
Temps	40 min	50 min	63 min	68 min	73 min	71 min
Surcoût	–	+ 10 min	+ 23 min	+ 28 min	+ 33 min	+ 31 min
Coûts pannes/sauvegardes			–	+ 5 min	+ 10 min	+ 8 min

TABLE 7.3 – Résultats des expériementations avec l'application de recherche phylogénique sur la plateforme HpcVm

Concernant les expériences avec l'application de recherche phylogénique, son temps moyen d'exécution de référence (avec les machines hôtes) est de 40 minutes. Pour cette application, son temps d'exécution en utilisant la plateforme, sans l'activation des mécanismes de gestion des demandes de sauvegarde est de 50 minutes, soit un surcoût de 10 minutes par rapport au temps de référence avec les machines hôtes. Pour les exécutions utilisant la plateforme avec les mécanisme de gestion des demandes de sauvegarde activés, le temps moyen d'exécution est de 63 minutes, soit un surcoût de 23 minutes par rapport au temps de référence sur les machines hôtes, et de 10 minutes par rapport au temps en utilisant la plateforme sans les mécanismes de gestion des demandes de sauvegarde. On peut remarquer que pour cette application, le surcoût induit par l'activation de ces mécanismes est légèrement plus élevé que pour la précédente application, avec un surcoût de 13 minutes. Ce coût légèrement plus élevé s'explique par la nature de l'application. Avec les données que nous utilisons, les itérations ne durent en moyenne que 20 secondes. Comme nous l'avons décrit dans la section 7.2.2, lorsqu'une tâche termine une itération elle émet une demande de sauvegarde auprès du démon qui la relaie au serveur. Si les conditions sont réunies, celui-ci déclenche le processus de sauvegarde, sinon il signifie son refus au démon qui indique à la tâche de continuer son exécution. Ces mécanismes ne représentent que quelques secondes, et chaque tâche est en attente de la réponse du démon. Cependant, ces quelques secondes ajoutées à la fin de chaque itération ont un impact non négligeable. Ceci montre que le choix de l'emplacement de l'appel aux mécanismes de sauvegarde doit être choisi judicieusement dans le code, afin de concilier performances et tolérance aux pannes. Un autre point important à souligner est le fait que les tâches de cette application sont indépendantes les unes des autres, et pour le fonctionnement des mécanismes de gestion des demandes de sauvegarde, toutes les tâches doivent se synchroniser. Ceci induit un ralentissement supplémentaire de l'application. L'impact d'une sauvegarde pour cette application est de 5 minutes et celui pour deux sauvegardes est de 10 minutes. On remarque ainsi que le coût induit par une sauvegarde est de 5 minutes. Pour cette application, le surcoût dû à une sauvegarde est légèrement moins élevé que pour la précédente application, d'environ 1 minute. Ceci s'explique par le fait que les tâches de cette application sont indépendantes et ne communiquent pas entre elles. Ainsi,

lors de la phase de transfert des sauvegardes, l'application n'est pas impactée par l'occupation de la bande passante. Enfin, concernant le contexte dans lequel nous n'effectuons qu'une sauvegarde et où nous introduisons une panne, le temps d'exécution de l'application est de 71 minutes, soit un surcoût de 8 minutes par rapport au temps d'exécution avec la plateforme ayant les mécanismes de gestion des demandes de sauvegarde activés. Comme le coût d'une sauvegarde est de 5 minutes, on peut en déduire que le coût d'une panne pour cette application est de 3 minutes, tout comme pour l'application précédente. Ceci confirme le fait que notre politique de tolérance aux pannes est efficace et a un surcoût assez faible.

Comme le montrent les résultats de nos expérimentations, l'architecture réseau sous-jacente entraîne des pertes de performances lorsque les tâches communiquent en utilisant un réseau logique en marge de la configuration mise en place. Comme nous l'avons expliqué dans la section 7.3.1, le réseau à notre disposition n'est pas favorable à la mise en place d'un réseau logique parallèle. Afin de nuancer les résultats de nos expérimentations, nous avons mené une étude comparative portant sur les transferts de fichiers. Pour cela, nous avons réaliser le transfert de fichiers de différentes tailles afin de mesurer les performances des liens de communication. Pour ce faire nous avons utilisé trois tailles de fichier : 10Mo, 100Mo, et 1Go que nous avons transférés entre : deux machines hôtes d'une même salle (Hs1-Hs1), deux machines hôtes de deux salles différentes (Hs1-Hs2), deux machines virtuelles dans la même salle (Vs1-Vs1), et deux machines virtuelles de deux salles différentes (Vs1-Vs2). Nous avons aussi mesuré la latence des liens communication entre les différentes machines. Les résultats de cette étude sont présentés dans le tableau 7.4.

Machines / Taille	Hs1 - Hs1	Hs1 - Hs2	Vs1 - Vs1	Vs1 - Vs2
10 Mo	<1 s	1 s	21 (<1) s	21 (1) s
100 Mo	1 s	10 s	23 (3) s	30 (10) s
1 Go	15 s	90 s	60 (40) s	130 (110) s
Latence	0,318 ms	0,503 ms	0,876 ms	1,164 ms

TABLE 7.4 – Étude comparative des performances des liens de communication

Concernant les transferts de fichiers entre les machines hôtes, on peut remarquer que les temps de transfert entre deux machines de deux salles différentes est bien plus élevé que celui entre deux machines d'une même salle. Ceci indique que même sans l'utilisation de notre plateforme de calcul, le fait d'utiliser des machines des deux salles pour l'exécution d'une application entraîne une perte de performances, notamment pour les applications dont les tâches communiquent entre elles. Concernant les transferts de fichiers entre les machines virtuelles, dans le tableau sont indiqués deux chiffres :

- le chiffre entre parenthèses représente le temps de transfert réel du fichier (tr). C'est-à-dire que ce temps permet de mesurer le surcoût induit par la virtualisation de la carte réseau ;
- le chiffre non parenthésé représente le temps total de transfert du fichier (tt). Dans ce temps sont compris le temps réel de transfert (tr) plus un surcoût dû à l'utilisation d'un réseau logique à part pour les machines virtuelles.

Les résultats montrent que les temps de transfert réels (tr) sont quasiment équivalent à ceux des machines hôtes, le surplus étant induit par la virtualisation de l'interface réseau. On remarque que le temps total (tt) associé est bien plus élevé. Ce temps comprend le temps de transfert, mais surtout les lenteurs induites par la configuration réseau de l'architecture. Une explication possible serait que les commutateurs « intelligents » fonctionnent comme des concentrateurs lorsqu'ils ne connaissent pas la

classe d'adresse utilisée (pour rappel, les machines virtuelles utilisent un réseau logique à part de celui des machines hôtes). Ainsi le surcoût serait dû à la recherche de routes entre les machines virtuelles. Ces observations se retrouvent dans l'évaluation de la latence entre les machines. On constate un léger écart entre la latence entre les machines d'une même salle et des machines de deux salles différentes. Le constat est identique pour les machines virtuelles, avec ici une nette augmentation dûe à la virtualisation de l'interface réseau.

Les résultats de cette étude montrent que la solution utilisée ici pour la configuration réseau n'est pas optimale. Elle permet de nuancer les résultats des expérimentations, donnés précédemment. Afin de pallier ces problèmes de réseau, dûs à l'usage d'un réseau logique parallèle, une nouvelle méthode de communication pour la plateforme HpcVm est à l'étude.

7.4 Conclusion

Dans ce chapitre nous avons présenté notre plateforme de calcul, nommée HpcVm, utilisant des machines virtuelles. L'utilisation de machines virtuelles permet de s'affranchir de la dépendance au matériel des machines hôtes, créant ainsi une plateforme d'exécution homogène. De plus, le fait d'utiliser des machines virtuelles permet une mise œuvre plus simple de politiques de tolérance aux pannes, ayant l'avantage d'être plus transparentes pour l'utilisateur final.

Dans un premier temps nous avons décrit les choix techniques que nous avons effectués afin de satisfaire nos objectifs. Comme nous l'avons vu, afin de faciliter l'exécution et le déploiement de la plateforme, nous avons choisi de l'écrire en langage Java. Ce langage a l'avantage d'être multi-platforme et permet une isolation des entités de la plateforme par rapport aux autres logiciels sur les machines hôtes. Ceci permet d'augmenter la sécurité des applications exécutées dans les machines virtuelles. Pour la technique de virtualisation, nous avons opté pour un hyperviseur de type 2, en utilisant VMWare Player. Ce type d'hyperviseur permet un déploiement rapide des machines virtuelles car celui-ci s'installe comme un logiciel classique, tout en offrant une bonne isolation des applications exécutées dans la machine virtuelle. Nous avons également décrit le choix technique que nous avons effectué pour les transferts des sauvegardes des machines virtuelles.

Nous avons ensuite présenté l'architecture de HpcVm en détaillant ses éléments constitutifs et les relations existantes entre eux. Puis nous avons décrit le fonctionnement général de la plateforme en présentant chacune de ses trois phases d'utilisation, allant de l'installation et la configuration de la plateforme à l'exécution d'une application, en passant par la phase de déploiement des machines virtuelles. Ensuite nous avons décrit les opérations nécessaires au déploiement de l'image de la machine virtuelle sur les machines hôtes, ainsi que celles dédiées à la mise en fonction des machines qui seront utilisées pour le calcul. Enfin nous avons vu le déroulement d'une exécution d'une application sur la plateforme. Nous avons aussi décrit le fonctionnement des mécanismes de sauvegarde et de tolérance aux pannes.

Enfin, nous avons présenté les expérimentations que nous avons menées afin d'évaluer les performances de notre plateforme. Pour cela nous avons utilisé les machines de travail destinées aux travaux pratiques des étudiants du département Informatique de l'IUT de Belfort-Montbéliard. Nous avons utilisé deux applications de calcul, l'une permettant la résolution de l'équation d'advection-diffusion et l'autre étant une application de recherche phylogénique. Les résultats de ces expérimentations sont encourageants quant à l'utilisation de notre plateforme et démontrent sa viabilité. Le point négatif concerne le rapport des temps induit par les mécanismes de sauvegarde. Cependant, les résultats présentés sont toutefois à relativiser en vertu de la taille des applications que nous avons utilisées. En effet, ces applications n'ont pas un temps d'exécution assez long, comme celles visées par notre étude, et les coûts fixes dûs aux mécanismes de sauvegarde et de tolérance aux pannes prennent une grande

importance. Sur des applications plus longues (plusieurs heures voire plusieurs jours) ces surcoûts ont un impact plus faible. De même, comme nous l'avons vu pour la seconde application, un placement judicieux de l'appel aux mécanismes de sauvegarde permettrait de limiter leur impact.

Comme on peut le remarquer, la plateforme HpcVm est opérationnelle, bien qu'elle soit encore à l'état de prototype. De nombreux points sont à améliorer comme la décentralisation du serveur, permettant ainsi une tolérance aux pannes complète de la plateforme et favorisant également son passage à l'échelle. Dans une version future, les objectifs sont de permettre l'utilisation d'autres systèmes d'exploitation au sein des machines virtuelles. De même, une couche virtuelle de communication entre les machines virtuelles est à l'étude, ceci afin de sécuriser et d'accroître les performances des communications entres les machines virtuelles. L'objectif de cette couche virtuelle de communication est de s'affranchir des difficultés et des pertes de performances occasionnées par l'intrusion d'un réseau annexe sur celui reliant les machines hôtes.

Conclusion et perspectives

Conclusion

Aujourd'hui, les applications scientifiques doivent traiter des données de plus en plus volumineuses. Les enjeux du calcul numérique parallèle se portent maintenant sur l'utilisation d'architectures distribuées comportant un grand nombre de machines, comme les grilles de calcul par exemple. Ce type d'architecture permet de fédérer les ressources, telles que des clusters ou des supercalculateurs, de diverses entités pouvant être géographiquement éloignées. Afin d'utiliser au mieux ces architectures, des méthodes de calcul ont été conçues avec pour objectif d'être efficaces et de gérer les problèmes induits par ces architectures. En effet, en contrepartie des nombreuses ressources qu'elles fournissent, ces architectures possèdent des inconvénients à prendre en compte lors de l'exécution d'une application. Les trois principaux inconvénients de ces architectures sont l'hétérogénéité des machines de calcul, la fréquence de leurs pannes, et l'hétérogénéité des liens de communication entre les différentes ressources. Nous avons vu que le modèle itératif asynchrone est bien adapté à ce type d'architecture en supprimant toute forme de synchronisation entre les tâches de calcul et en étant tolérant à la perte de messages de données. Néanmoins, un algorithme itératif asynchrone progresse plus rapidement si les tâches reçoivent régulièrement des messages de dépendances, et évidemment si les machines sur lesquelles elles sont exécutées leur permettent de calculer plus rapidement. Ainsi, lors de l'exécution d'une application itérative asynchrone sur une plateforme de calcul fournissant de nombreuses ressources, il est crucial de judicieusement sélectionner les machines qui seront utilisées pour effectuer les calculs.

Dans la partie II de cette thèse, nous nous sommes intéressés à cette problématique. Ses enjeux se portent sur un choix dans lequel interviennent divers paramètres pouvant être antagonistes. En effet, un algorithme de placement doit à la fois tenir compte de la localité des tâches et choisir les machines permettant de terminer au plus tôt l'application, tout en prenant en compte la possibilité de l'apparition de pannes. Dans le chapitre 3 nous avons décrit les enjeux de cette problématique et nous avons défini les modélisations et notions associées. Le principal enjeu de nos travaux est de fournir un algorithme de placement qui prend en compte à la fois la localité des tâches de calcul et la puissance des machines, sans négliger la tolérance aux pannes. De même, ce placement doit tenir compte de l'architecture d'exécution mais également des caractéristiques des applications, comme la charge de calcul des tâches et de leurs dépendances. Dans le chapitre 4 nous avons présenté trois algorithmes dédiés au placement des tâches des applications itératives asynchrones sur des architectures distribuées, hétérogènes et volatiles. Les deux premiers algorithmes que nous avons présentés sont des adaptations d'algorithmes de la littérature à la résolution de notre problématique. Nous avons conçu le troisième algorithme en ayant pour objectif de prendre en compte tous les aspects de notre problématique. Ces

trois algorithmes sont :

- FT-FEC, dont l'objectif est d'optimiser la localité des tâches de calcul en évitant d'utiliser les liens de communication pénalisants ;
- FT-AIAC-QM, dont l'objectif est de réduire le temps d'exécution des tâches de calcul ;
- MAHEVE, qui s'inspire des meilleurs aspects des deux précédents, en s'adaptant tant aux caractéristiques de l'architecture d'exécution qu'à celles des applications.

Afin de satisfaire nos objectifs, chacun de ces algorithmes de placement possède une fonction permettant de sélectionner une machine de remplacement lorsqu'une machine exécutant une tâche de l'application tombe en panne. Ces trois algorithmes ont été intégrés à l'environnement JaceP2P-V2 à l'aide d'une bibliothèque, appelée *Mapping*. Cet environnement permet de développer et d'exécuter des applications itératives asynchrones sur des architectures distribuées, hétérogènes et volatiles – c'est le seul environnement existant dans ce domaine dont nous avons la connaissance.

Dans le chapitre 5 nous avons présenté les expérimentations que nous avons menées afin de vérifier l'efficacité de nos algorithmes de placement. Nous avons réalisé des expériences en utilisant des architectures et des applications réelles, car la nature non prévisible du modèle itératif asynchrone rend très difficile l'utilisation de simulations. Nous avons utilisé deux applications itératives asynchrones, l'une étant la multidécomposition avec gradient conjugué et l'autre permettant la résolution de l'équation d'avection-diffusion tridimensionnelle. Pour chaque application, nous avons utilisé trois types d'architecture d'exécution, sélectionnées parmi les ressources de la grille d'expérimentation Grid'5000 : une architecture homogène, une architecture hétérogène, et une architecture neutre. Ces trois types d'architecture permettent d'observer le comportement de nos trois algorithmes suivant le degré d'hétérogénéité de l'architecture d'exécution. Enfin, pour chaque application et chaque architecture, nous avons exécuté les applications avec et sans l'introduction de pannes durant les calculs. Les résultats montrent que nos trois algorithmes permettent de réduire significativement le temps d'exécution des applications, et ce dans tous les cas de figure que nous avons utilisés. Les deux algorithmes FT-FEC et FT-AIAC-QM sont efficaces respectivement sur les architectures homogènes et sur celles hétérogènes. Notre algorithme MAHEVE est le plus efficace sur les trois types d'architecture, et propose la meilleure politique de tolérance aux pannes, en réduisant le plus les coûts induits par l'introduction de pannes au cours des exécutions des applications.

Dans la partie III, nous avons souligné le fait que tous les scientifiques et industriels ne peuvent pas forcément avoir accès à une architecture du type grille de calcul pour exécuter leurs applications. Généralement, les temps d'exécution de ces applications sont très longs, pouvant durer plusieurs jours. Nous proposons une alternative consistant à utiliser les machines disponibles au sein de leur propre institution, telles que les entreprises ou les universités. L'utilisation de ces machines durant leur temps d'inactivité permet aux utilisateurs d'effectuer des calculs sans avoir recours à des solutions telles que le « cloud computing » par exemple, impliquant l'externalisation des données et des processus pouvant être confidentiels. Dans le chapitre 6 nous présentons les motivations et les principes de l'utilisation de ces machines internes aux institutions, à l'aide de plateformes utilisant des machines virtuelles. La virtualisation procure de nombreux avantages, comme la préservation de la confidentialité des données, la simplicité d'utilisation tant pour leur mise en œuvre que pour l'exécution des applications. De plus elle offre des possibilités pour assurer la terminaison des applications et ce même si des pannes surviennent, grâce à l'ajout de mécanismes de tolérance aux pannes. Nous avons défini les objectifs d'une telle plateforme : les utilisateurs ciblés étant généralement non informaticiens, elle doit être simple à installer et à utiliser, et doit assurer la terminaison des applications, même si leur temps d'exécution est très long (de quelques heures à plusieurs jours). De plus, son utilisation doit nécessiter le moins d'intervention possible de l'utilisateur sur le code de ses applications. L'utilisation de machines virtuelles permet de s'affranchir de l'hétérogénéité des machines hôtes, simplifiant également la tâche de l'utilisateur qui ne doit alors préparer son application que pour un type de machine, celui de la

machine virtuelle choisie. Après avoir décrit les différentes techniques de virtualisation existantes, nous avons présenté deux plateformes de calcul utilisant des machines virtuelles, *i*Share et Harmony. Ces deux plateformes, bien que performantes dans leurs domaines respectifs ne satisfont pas nos objectifs.

Dans le chapitre 7, nous avons présenté la mise en œuvre de notre prototype de plateforme de calcul pour les architectures de type « enterprise desktop grid » en utilisant des machines virtuelles, nommée HpcVm. Nous avons tout d'abord présenté les choix techniques que nous avons effectués afin de permettre la réalisation de nos objectifs. Ensuite nous avons décrit ses trois éléments principaux (le serveur, les démons et l'interface utilisateur) permettant la gestion des machines virtuelles sur les machines hôtes. Nous avons vu que l'installation et la configuration de cette plateforme sont simplifiées grâce à l'utilisation de logiciels multiplateformes. Nous avons également décrit ses mécanismes de sauvegarde et de tolérance aux pannes, lui permettant d'assurer la terminaison des applications. Ces mécanismes ne nécessitent que très peu de modifications au sein du code des applications, rendant ainsi plus facile l'usage de notre plateforme. Les mécanismes de sauvegarde et de tolérance aux pannes reposent sur la méthode de sauvegarde/restauration de l'état des machines virtuelles, rendu possible grâce à l'utilisation de la technique de virtualisation utilisant un hyperviseur de type 2. Cette technique permet d'exécuter une machine virtuelle comme un programme classique, tout en offrant une bonne isolation des applications et des calculs s'y déroulant, préservant ainsi la confidentialité des données et des processus. Nous avons également présenté les résultats des expérimentations que nous avons menées afin d'évaluer les performances de notre plateforme de calcul. Nous avons utilisé deux applications, l'une permettant la résolution de l'équation d'advection-diffusion et l'autre étant une application de recherche phylogénique. Les résultats ont montré que bien qu'étant encore à l'état de prototype, notre plateforme permet d'atteindre les objectifs que nous nous sommes fixés. En effet, elle permet d'assurer la terminaison des applications même lorsque les machines hôtes tombent en panne. Nous avons également vu que le surcoût induit par l'utilisation des machines virtuelles reste très acceptable, si l'on se replace dans le contexte visé. En effet, pour des applications requérant de longs temps d'exécution (plusieurs jours), en effectuant des sauvegardes toutes les deux heures par exemple, les surcoûts deviennent moins importants.

Perspectives

Dans cette thèse nous avons présenté nos travaux portant dans une première partie sur des algorithmes de placement des tâches d'applications itératives asynchrones sur des architectures hétérogènes et volatiles, et dans une seconde partie sur la mise en œuvre d'une plateforme de calcul utilisant des machines virtuelles. Bien que présentant de bons résultats, plusieurs pistes d'améliorations se présentent, pour chacune de nos contributions.

Algorithmes de placement

Comme nous l'avons vu, les algorithmes de placement que nous avons présentés permettent de réduire le temps d'exécution des applications itératives asynchrones, ainsi que des fonctions de tolérance aux pannes efficaces. L'algorithme proposant les meilleures performances est notre algorithme MAHEVE. Cependant, nous souhaitons poursuivre les recherches sur ces algorithmes. Nos principaux axes de recherche sont :

- mener des expérimentations supplémentaires avec d'autres applications afin de valider les performances que nous avons obtenues. En effet, comme nous avons pu nous en apercevoir, les particularités du modèle itératif asynchrone font que chaque application peut adopter un comportement différent. Le fait d'exécuter d'autres applications peut nous indiquer quels sont les

points faibles de nos algorithmes et ainsi les améliorer en conséquence ;

- pour l'algorithme MAHEVE ajouter une gestion plus fine des caractéristiques de l'architecture d'exécution et des applications. Il serait judicieux de tenir compte de la quantité de mémoire disponible afin de choisir plus finement les machines de calcul. En effet, lors du choix de nos architectures de calcul dans nos expérimentations, certaines machines ont été exclues « manuellement » car elle ne possédaient pas assez de mémoire pour exécuter nos applications ;

- ajouter la prise en compte du débit des liens de communication dans l'algorithme MAHEVE. Dans sa version actuelle, l'algorithme n'utilise que la latence des liens, or il pourrait être intéressant d'intégrer le débit de ces derniers, surtout lorsque les tâches des applications échangent des données volumineuses ;

- comme nous avons pu le remarquer lors des expériences avec l'application résolvant l'équation d'avection-diffusion tridimensionnelle, lorsque les tâches d'une application possèdent de nombreuses dépendances et que celles-ci échangent un volume important de données, les algorithmes de placement doivent fortement en tenir compte. Ainsi, nous souhaiterions inclure dans notre stratégie de placement une plus grande prise en compte des caractéristiques des applications, comme par exemple le nombre de dépendances des tâches ;

- ajouter une fonction aux algorithmes afin qu'ils puissent indiquer à l'environnement d'exécution, notamment JaceP2P-V2, les machines sur lesquelles les tâches doivent effectuer leur sauvegarde. Actuellement, les sauvegardes sont envoyées sur les voisins de calcul des tâches. Avec les politiques de placement que nous utilisons, ces voisins sont généralement situés au sein d'un même cluster. Le problème vient du fait que si tout un cluster tombe en panne, tous les calculs sont perdus et l'utilisateur doit relancer l'application. Le fait de mettre en œuvre une telle fonction permettrait d'accroître le degré de tolérance aux pannes de l'environnement ;

- améliorer certaines fonctionnalités de la bibliothèque « Mapping », comme par exemple la création du graphe d'architecture. Dans sa version actuelle, lors de la création du graphe d'architecture, la formation des clusters est basée sur le nom des machines, principalement dans le style Grid'5000 (par exemple cluster-numéro.site.grid5000.fr). Cette méthode pourrait être généralisée en utilisant la performance des liens de communication entre les machines pour définir les clusters.

Plateforme de calcul HpcVm

Comme nous l'avons indiqué dans la partie III, notre plateforme HpcVm, bien que fonctionnelle, n'est encore qu'un prototype. De nombreuses pistes d'améliorations sont possibles. Parmi celles-ci, les plus intéressantes sont :

- une amélioration essentielle serait de réaliser la décentralisation de la plateforme. En effet, comme nous l'avons indiqué, le serveur est le point central supportant toutes les charges, comme la surveillance des démons, la gestion de la tolérance aux pannes, le stockage et le déploiement des machines virtuelles. Une première modification serait de déporter la surveillance des démons. Ceci peut être réalisé de la même manière que pour la plateforme JaceP2P-V2, en utilisant un mécanisme où chaque démon en surveille un autre et avertit l'entité de supervision de cette panne. De même, lorsqu'une application est en cours d'exécution, le serveur continue de surveiller les démons participant à cette exécution. Une solution pour délester le serveur de cette tâche serait d'ajouter à la plateforme une entité « lanceur » qui serait en charge d'assurer le bon déroule-

ment de l'application en supervisant les machines qui participent à l'exécution de son application ;

- une amélioration possible serait d'étudier les possibilités d'utiliser des méthodes de sauvegardes incrémentales des machines virtuelles. En effet, actuellement lors d'une opération sauvegarde, une archive de la machine virtuelle dans sa totalité est créée puis transférée vers une autre machine hôte. Le fait d'utiliser des sauvegardes incrémentales permettrait de réduire le temps de sauvegarde et surtout de limiter l'usage de la bande passante lors des transferts. De même, un transfert moins long permet de valider plus rapidement une étape de sauvegarde ;

- une autre amélioration serait la conception d'un mécanisme de communication permettant de s'affranchir de l'infrastructure réseau reliant les machines hôtes. En effet, comme nous l'avons indiqué, nous utilisons actuellement un réseau logique parallèle. Un tel mécanisme, même s'il est toléré par la configuration du réseau hôte entraîne des pertes de performances au niveau des communications entre les tâches de calcul. L'idée serait de mettre en place un réseau virtuel entre les machines virtuelles, en se basant par exemple sur une méthode de table de hachage distribuée pour la gestion des routes (la clé étant l'adresse IP de la machine virtuelle et le champ l'adresse IP de l'hôte). Les démons seraient en charge d'intercepter les communications et de les transférer directement à leurs destinataires, en empruntant les mécanismes de routage du réseau hôte ;

- enfin, une dernière amélioration envisageable serait l'ajout d'un mécanisme permettant de sélectionner plus finement les machines qui seront utilisées pour déployer les machines virtuelles. En effet, dans une architecture de type « enterprise desktop grid » les machines sont hétérogènes, fournissant ainsi des ressources inégales (processeurs, quantité de mémoire...). L'idée serait de permettre une sélection des machines hôtes en fonction des caractéristiques des machines virtuelles, et plus particulièrement de l'espace mémoire qui lui est alloué. En effet, il est plus efficace de pouvoir loger cet espace dans la mémoire de la machine hôte. De ce fait un tel mécanisme de sélection permettrait de choisir les machines hôtes disponibles possédant suffisamment de mémoire pour permettre une exécution des machines virtuelles dans de bonnes conditions ;

- une dernière perspective viserait à la mise en place de mécanismes de détection proactive des défaillances des machines hôtes. Ceci permettrait d'utiliser des fonctionnalités de « live migration » (migration en direct) des machines virtuelles. Un tel mécanisme permet de ne pas arrêter les calculs en cours afin de redémarrer les sauvegardes, et donc les calculs. Ceci permettrait de gagner en performances et de limiter l'impact des pannes sur le temps d'exécution des applications.

Publications

Journaux internationaux

Conférences internationales

[C1] Nabil Abdennadher, Mohamed Ben Belgacem, Raphaël Couturier, David Laiymani, Sébastien Miquée, Marko Niinimaki, and Marc Sauget. Gridification of a radiotherapy dose computation application with the XtremWeb-CH environment. In Jukka Riekki, Mika Ylianttila, and Minyi Guo, editors, *Advances in Grid and Pervasive Computing*, volume 6646 of *LNCS*, pages 188–197. Springer Berlin / Heidelberg, Oulu, Finland, 2011.

[C2] Raphaël Couturier, David Laiymani, and Sébastien Miquée. MAHEVE : An efficient reliable mapping of asynchronous iterative applications on volatile and heterogeneous environments. In *HeteroPar'10, 8-th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, volume 6586 of *LNCS*, pages 31–39, Ischia, Italy, August 2010. Springer.

[C3] Raphaël Couturier, David Laiymani, and Sébastien Miquée. Mapping asynchronous iterative applications on heterogeneous distributed architectures. In *IPDPS'10, ACM/IEEE Int. Parallel and Distributed Processing Symposium, Workshop on Parallel and Distributed Scientific and Engineering Computing*, Atlanta, USA, 2010. IEEE Computer Society Press.

[C4] Raphaël Couturier, David Laiymani, and Sébastien Miquée. High performance computing using ProActive environment and the asynchronous iteration model. In *IPDPS'09, ACM/IEEE Int. Parallel and Distributed Processing Symposium, Workshop on Java and Components for Parallelism, Distribution and Concurrency*, Rome, Italy, 2009. IEEE Computer Society Press.

Rapport de stage de fin de cycle ingénieur

[R1] S. Miquée. Étude comparative des environnements de calcul haute performance Jace et Proactive, 2008. Rapport Stage Ingénieur/Master.

Bibliographie

[1] Folding@home. `http://folding.stanford.edu`.

[2] Grid'5000. `http://www.grid5000.fr`.

[3] Raxml application. `http://www.exelixis-lab.org/`.

[4] Réseau RENATER. `http://www.renater.fr`.

[5] Vmware – virtual infrastructure software. `http://www.vmware.com`.

[6] D. L. Long L. A. and Clarke. Task interaction graph : An intermediate representation for concurrency. Technical report, University of Massachusetts, Amherst, MA, USA, 1988.

[7] N. Abdennadher and R. Boesch. Towards a peer-to-peer platform for high performance computing. In *Advances in Grid and Pervasive Computing, Second International Conference, GPC 2007, Paris, France, May 2-4, 2007, Proceedings*, volume 4459 of *Lecture Notes in Computer Science*, pages 412–423. Springer, 2007.

[8] Nabil Abdennadher, Mohamed Ben Belgacem, Raphaël Couturier, David Laiymani, Sébastien Miquée, Marko Niinimäki, and Marc Sauget. Gridification of a radiotherapy dose computation application with the XtremWeb-CH environment. In *GPC 2011, Grid and Pervasive Computing*, volume 6646 of *LNCS*, pages 188–197, Oulu, Finland, May 2011.

[9] L. Alvisi and K. Marzullo. Message logging : Pessimistic, optimistic, and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 229–236, British Columbia, Canada, 1995. IEEE Computer Society press.

[10] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home : An experiment in public-resource computing. *Communications of the ACM (CACM)*, 45(11) :56–61, November 2002.

[11] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Qu ilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[12] J. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing AIAC algorithms. *Journal of Supercomputing*, 35(3) :227–244, 2006.

[13] J. Bahi, S. Contassot-Vivier, and R. Couturier. *Parallel Iterative Algorithms : from Sequential to Grid Computing*, volume 1 of *Numerical Analysis & Scientific Computating*, chapter Asynchronous Iterations, pages 124–131. Chapman & Hall/CRC, 2007.

[14] J. Bahi, R. Couturier, and P. Vuillemin. JaceP2P : an environment for asynchronous computations on peer-to-peer networks. In *Cluster 2006, IEEE Int. Conf. on Cluster Computing*, pages 1–10. IEEE Computer Society Press, 2006.

[15] J. Bahi, S. Domas, and K. Mazouzi. Jace : a java environment for distributed asynchronous iterative computations. In *12th Euromicro Conference on Parallel, Distributed and Network based Processing, PDP'04*, pages 350–357, Coruna, Spain, February 2004. IEEE computer society press.

[16] J. M. Bahi, S. Contassot-Vivier, L. Makovicka, E. Martin, and M. Sauget. Neurad. *Agence pour la Protection des Programmes. No : IDDN.FR.001.130035.000.S.P.2006.000.10000*, 2006.

[17] J. M. Bahi, R. Couturier, K. Mazouzi, and M. Salomon. Synchronous and asynchronous solution of a 3D transport model in a grid computing environment. *Applied Mathematical Modelling*, 2005.

[18] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Paralell Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[19] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[20] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. Mpich-v : a multiprotocol fault tolerant mpi. *In International Journal of High Performance Computing and Applications*, 20(3) :319–333, 2006.

[21] Nicholas S. Bowen, Christos Nikolaou, and Arif Ghafoor. On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems. *IEEE Trans. Computers*, 41(3) :257–273, 1992.

[22] G. Cao and M. Singhal. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, PDS-9(12) :1213–1225, December 1998.

[23] J.-C. Charr, R. Couturier, and D. Laiymani. JaceP2P-V2 : A fully decentralized and fault tolerant environment for executing parallel iterative asynchronous applications on volatile distributed architectures. In *GPC*, pages 446–458, 2009.

[24] R. Couturier, D. Laiymani, and S. Miquée. High performance computing using proactive environment and the asynchronous iteration model. In *IPDPS'09, ACM/IEEE Int. Parallel and Distributed Processing Symposium, Workshop on Java and Components for Parallelism, Distribution and Concurrency*, Rome, Italy, 2009. IEEE Computer Society Press. Proceedings on CD-ROM.

[25] Christophe Cérin and Gilles Fedak. *Desktop Grid Computing.* Numerical Analysis & Scientific Computating. Chapman & Hall/CRC, 2012.

[26] Jeff Dike. User-mode Linux. In *ALS'01 : Proceedings of the 5th annual conference on Linux Showcase & Conference*, page 2, Berkeley, CA, USA, 2001. USENIX Association.

[27] Patricio Domingues, Filipe Araujo, and Luis Silva. Evaluating the performance and intrusiveness of virtual machines for desktop grid computing. volume 0, pages 1–8, Los Alamitos, CA, USA, May 2009. IEEE Computer Society.

[28] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3) :375–408, 2002.

[29] C. Farhat. A simple and efficient automatic fem domain decomposer. *Computers & Structures*, 28(5) :579 – 602, 1988.

[30] G. Fedak, C. Germain, V. Néri, and F. Cappello. Xtremweb : A generic global computing system. In *International Symposium on Cluster Computing and the Grid*, pages 582–587. IEEE Computer Society, 2001.

[31] P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *DOA*, pages 132–141, 1999.

[32] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, Summer 1997.

[33] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI : Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[34] M. Garey and D. Johnson. *Computer and Intractability : a guide to the Theory of NP-Completness.* W.H. Freeman & Co, 1979.

[35] Stéphane Genaud, Emmanuel Jeannot, and Choopan Rattanapoka. Fault-Management in P2P-MPI. *International Journal of Parallel Programming*, 37(5) :433–461, 2009.

[36] Stéphane Genaud and Choopan Rattanapoka. P2P-MPI : A peer-to-peer framework for robust execution of message passing parallel programs on grids. *J. Grid Comput.*, 5(1) :27–42, 2007.

[37] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standar d. *Parallel Computing*, 22(6) :789–828, September 1996.

[38] B. Hendrickson and R. W. Leland. *The Chaco User's Guide.* Sandia National Laboratory, Albuquerque, 1995.

[39] A. C. Hindmarsh. User documentation for PVODE, an ODE solver for parallel computers, 2007.

[40] S.R.H. Hoole. Optimal design, inverse problems and parallel computers. *IEEE Transactions on Magentics*, 27(5) :4146–4149, September 1991.

[41] S. Huang, E. E. Aubanel, and V. C. Bhavsar. Pagrid : A mesh partitioner for computational grids. *J. Grid Comput.*, 4(1) :71–88, 2006.

[42] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1) :359–392, 1998.

[43] Israel Koren and C. Mani Krishna. *Fault Tolerant Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[44] S. Kumar, S. K. Das, and Rupak Biswas. Graph partitioning for parallel applications in heterogeneous grid environments. In *IPDPS*, 2002.

[45] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5) :506–521, 1996.

[46] E. Laure, C. Gr, S. Fisher, A. Frohner, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, R. Byrom, L. Cornwall, M. Craig, A. Di Meglio, A. Djaoui, F. Giacomini, J. Hahkala, F. Hemmer, S. Hicks, A. Edlund, A. Maraschini, R. Middleton, M. Sgaravatto, M. Steenbakkers, J. Walk, and A. Wilson. Programming the Grid with gLite. In *Computational Methods in Science and Technology*, volume 12, 2006.

[47] Siming Lin and Xueqi Cheng. BC-GA : A graph partitioning algorithm for parallel simulation of internet applications. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0 :358–365, 2008.

[48] D. L. Long and L. A. Clarke. Task interaction graphs for concurrency analysis. In *ICSE*, pages 44–52, 1989.

[49] Niloufer Mackey. Hamilton and Jacobi Meet Again : Quaternions and the Eigenvalue Problem. *SIAM Journal on Matrix Analysis and Applications*, 16(2) :421–435, 1995.

[50] MPI Forum. MPI : A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883, Portland, OR, November 1993. IEEE CS Press.

[51] Vijay K. Naik, Swaminathan Sivasubramanian, David Bantz, and Sriram Krishnan. Harmony : A desktop grid for delivering enterprise computations. In *Fourth International Workshop on Grid Computing (GRID'03)*, 2003.

[52] Zhelong Pan, Rudolf Eigenmann, and Dongyan Xu. Executing MPI Programs on Virtual Machines in an Internet Sharing System. In *In Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS-2006)*, 2006.

[53] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Linpack Benchmark. `http://www.netlib.org/benchmark/hpl/index.html`.

[54] P. Phinjaroenphan. *An Efficient, Pratical, Portable Mapping Technique on Computational Grids.* PhD thesis, School of Computer Science and Information technology Science, Engineering and Technology Portfolio, RMIT University, 2006.

[55] P. Phinjaroenphan, S. Bevinakoppa, and P. Zeephongsekul. A heuristic algorithm for mapping parallel applications on computational grids. In *EGC*, pages 1086–1096, 2005.

[56] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt : Transparent checkpointing under UNIX. In *USENIX Winter*, pages 213–224, 1995.

[57] J. K. Reid. *On the method of conjugate gradients for the solution of large sparse systems of linear equations*, pages 231–254. Academic Press Inc, March 1971.

[58] X. Ren and R. Eigenmann. ishare - open internet sharing built on peer-to-peer and web. In *In European Grid Conference, Amsterdam,TheNetherlands,Feb*, 2005.

[59] R. Van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection. In *Service,ž01d Proc. Conf. Middleware*, pages 55–70, 1998.

[60] Youcef Saad and Martin H. Schultz. GMRES : a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3) :856–869, July 1986.

[61] S. Sanyal, A. Jain, S. K. Das, and Rupak Biswas. A hierarchical and distributed approach for mapping large applications to heterogeneous grids using genetic algorithms. In *CLUSTER*, pages 496–499, 2003.

[62] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, USA, 1989.

[63] Alexandros Stamatakis. RAxML-VI-HPC : maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics (Oxford, England)*, 22(21) :2688–2690, November 2006.

[64] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice : the condor experience. *Concurrency - Practice and Experience*, 17(2-4) :323–356, 2005.

[65] H. Topcuouglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3) :260–274, 2002.

[66] J. G. Verwer, J. G. Blom, and W. Hundsdorfer. An implicit-explicit approach for atmospheric transport-chemistry problems. *Applied Numerical Mathematics : Transactions of IMACS*, 20(1–2) :191–209, February 1996.

[67] G. von Laszewski, M. Parashar, A.G. Mohamed, and G.C. Fox. On the parallelization of blocked lu factorization algorithms on distributed memory architectures. In *Supercomputing*, pages 170–179, November 1992.

[68] Jon B. Weissman and Andrew S. Grimshaw. A framework for partitioning parallel computations in heterogeneous environments. *Concurrency - Practice and Experience*, 7(5) :455–478, 1995.

[69] Jinchao Xu. Iterative Methods by Space Decomposition and Subspace Correction. *SIAM Review*, 34(4) :581–613, 1992.

[70] T. Yang and A. Gerasoulis. Dsc : Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9) :951–967, 1994.

Résumé

La technologie actuelle permet aux scientifiques de divers domaines d'obtenir des données de plus en plus précises et volumineuses. Afin de résoudre ces problèmes associés à l'obtention de ces données, les architectures de calcul évoluent, en fournissant toujours plus de ressources, notamment grâce à des machines plus puissantes et à leur mutualisation. Ainsi, ces nouvelles architectures, telles que le regroupement des clusters en grilles de calcul par exemple, introduisent des problèmes d'hétérogénéité, de disponibilité, et de tolérance aux pannes. Afin d'exploiter au mieux ces architectures, le modèle itératif asynchrone offre de bonnes performances en permettant le recouvrement des communications par du calcul et en tolérant la perte de messages de données. Les applications sont découpées en tâches de calcul réparties sur les machines de l'architecture. Le choix de ces machines est un enjeu crucial.

Dans cette thèse, nous proposons d'étudier dans un premier temps le placement des tâches d'applications itératives asynchrones dans des environnements hétérogènes et volatils. Nous présentons les enjeux de ce placement et nous proposons trois algorithmes de placement dédiés à cette problématique. Les expérimentations que nous avons menées montrent qu'un placement efficace de ces tâches ainsi qu'une bonne politique de tolérance aux pannes permettent de réduire significativement les temps d'exécution de ces applications, et ce quel que soit l'architecture distribuée utilisée. Dans un second temps nous présentons la mise en œuvre du prototype d'une plateforme de calcul utilisant des machines virtuelles. L'objectif est d'utiliser les ressources inutilisées et/ou sous-exploitées au sein des institutions tout en ne modifiant que très peu les codes de calcul. Nous montrons que l'utilisation de machines virtuelles permet de répondre à nos objectifs. Notre solution permet également de s'affranchir de l'hétérogénéité des machines hôtes tout en offrant une implantation facilitée de politiques de tolérance aux pannes. Les expérimentations que nous avons menées sont encourageantes et montrent qu'il existe un réel potentiel quant à l'utilisation d'une telle plateforme pour l'exécution d'applications scientifiques.

Mots clefs : algorithmes parallèles itératifs asynchrones, hétérogénéité, algorithmes de placement, tolérance aux pannes, plateforme de calcul, machines virtuelles.

Abstract

The current technology allows scientists of several domains to obtain more precise and large data. In the same time, computing architectures evolve too, by providing even more computing resources, with more powerful machines and the pooling of them. So, these new architectures, like the gathering of clusters into computing grids for example, introduce heterogeneity, availability, and fault tolerance problems. In order to efficiently use these architectures, the asynchronous iteration model offers good performance by allowing the overlapping of communications by computations and it tolerates data messages loss. Applications are split into several computing tasks spread over architecture computing nodes. Choosing these nodes is a critical issue.

In this thesis, in a first time we propose to study the problem of the mapping of asynchronous iterative applications tasks into heterogeneous and volatile environments. We describe this mapping issue and we propose three mapping algorithms dedicated to it. Experimentations we conducted show that an efficient tasks mapping with a good fault tolerance policy allow to significantly reduce applications execution time, and this regardless of the distributed architecture used. In a second time we present the implementation of a computation platform using virtual machines. Our aim is to use unused and/or underutilized resources of institutions with little modifications into computing codes. We show that the use of virtual machines can meet our goals. Our solution allows also to overcome the heterogeneity of host machines while offering an easier implementation of policies for fault tolerance. The experiments we have conducted are encouraging and show that there is real potential for the use of such a platform for running scientific applications.

Key words : parallel asynchronous iterative algorithms, heterogeneity, mapping algorithms, fault tolerance, computing platform, virtual machines.