Environnements d'exécution pour applications parallèles communiquant par passage de messages pour les systèmes à grande échelle et les grilles de calcul

Camille Coti

Université Paris Sud - XI
Faculté de sciences d'Orsay
École doctorale d'Informatique
Numéro d'ordre : 9610

Thèse
Présentée pour l'obtention du titre de
Docteur de l'Université Paris Sud - XI

par Camille Coti

ENVIRONNEMENTS D'EXÉCUTION POUR APPLICATIONS PARALLÈLES COMMUNIQUANT PAR PASSAGE DE MESSAGES POUR LES SYSTÈMES À GRANDE ÉCHELLE ET LES GRILLES DE CALCUL

Soutenue le 10 novembre 2009

Président :	Yannis Manoussakis
Rapporteurs :	Jean-François Méhaut
	Raymond Namyst
Examinateurs :	Franck Cappello (directeur de thèse)
	Thomas Hérault (co-directeur de thèse)
	George Bosilca

Remerciements

Je remercie tout d'abord Thomas Hérault pour la confiance qu'il m'a accordée en m'offrant l'opportunité de préparer cette thèse dans un contexte large et enrichissant tel que le projet QosCosGrid ainsi qu'en m'offrant la chance d'effectuer une partie de mes travaux dans un laboratoire majeur du calcul à hautes performances, pour la liberté qu'il m'a laissée dans l'évolution de mes travaux, ainsi que pour l'attention qu'il a portée à la phase de rédaction de cette thèse. Je remercie également Franck Cappello qui a accepté de m'accueillir dans son équipe et de superviser le déroulement de ma thèse tout en acceptant de signer sans condition toutes mes demandes d'autorisation de cumul pour de l'enseignement sans râler ou presque (mais pas après moi).

Je tiens à remercier Jean-François Méhaut et Raymond Namyst pour avoir accepté d'être les rapporteurs de ma thèse. Le respect que j'ai pour leurs travaux sur les supports exécutifs pour les applications parallèles fait que c'est un honneur pour moi d'avoir soumis mes travaux à leur évaluation. Je les remercie pour les remarques constructives qu'ils ont faites sur le contenu de ma thèse, l'ouverture qu'a apporté leur point de vue et les corrections syntaxico-othographiques qu'ils m'ont suggérées. Je les remercie également pour leur disponibilité quant à leur participation à ma soutenance, sachant qu'ils étaient eux-mêmes soumis à beaucoup de contraintes. Je remercie également Yannis Manoussakis d'avoir accepté de présider le jury de la soutenance.

Je remercie George Bosilca d'avoir accepté de faire partie de mon jury et de s'être déplacé de loin spécialement pour assister à ma soutenance. Je le remercie également de m'avoir accueillie à trois reprises, pour trois longs séjours dans son équipe à Knoxville. Le soutien qu'il m'a apporté sur place (notamment en m'hébergeant à plusieurs reprises), sa générosité et son amitié me font le considérer comme plus qu'un encadrant. Ces remerciements doivent également être adressés à Gaëlle, tenancière de l'Auberge Bosilca.

Je remercie particulièrement Sylvain Peyronnet, qui m'a aidée à préparer ma soutenance de thèse suivant de multiples aspects. Vincent Neri, pour les batailles d'objets divers et variés (disques durs, savons, Yotta-Yottas...) et les discussions plus sérieuses. Sébastien Tixeuil, qui m'a généreusement proposé de recopier l'introduction de sa thèse pour m'éviter d'avoir à en écrire une moi-même. Joël Falcou, qui partage au labo les dragées du baptème de son fils et les descriptions des maladies de celui-ci. Pierre Lemarinier, qui a accepté de m'initier aux preuves d'algorithmes distribués et à la boxe française. William Hoarau et son rhum avec des vrais fruits qui marinent dedans (mais comment il a fait le litchi pour entrer dans la bouteille ?). Ben Quétier, pour son esprit de camaraderie communicatif. François Berenger, pour les discussions de g33k et d'autres choses, son soutien, et son amitié qui a dépassé les limites du labo.

Je remercie ceux qui ont été à mes côtés tout au long de ma thèse, avec qui nous nous sommes soutenus les uns les autres, Ala Rezmerita qui était dans l'équipe avant que j'arrive et y est toujours maintenant que j'en suis partie (tu vas finir par faire partie des meubles), et Paul Malécot, le petit marin Breton frisé qui, à l'heure où j'écris ces lignes, vient de soutenir sa thèse. Fatiha Bouabache et Olivier Pérès, mes "frère et sœur" scientifiques.

Je remercie les deux "filles d'Urbana", Amina Guermouche et Élisabeth Brunet, qui ont apporté chacune à sa façon leur bonne humeur au labo durant la fin de ma thèse. Aminette pour sa bonne humeur, son optimisme (mais si !) et son sourire qui remonte le moral. Babeth, à qui je n'en veux presque pas d'avoir été classée devant moi (je te dois toujours un grec !).

Également ceux qui étaient là quand je suis arrivée, comme Laurence Pilard qui a passé beaucoup de temps à m'expliquer le fonctionnement des choses, Éric Rodriguez, entre sa bonne humeur et ses colères devant les titres des journaux, ou qui sont arrivés vers la fin de ma thèse, comme Simplice Donfack, Amal Khabou qui a subi mes jeux de mots pourris, Guy-Antoine Atenekeng-Kahou, pointu sur la grammaire et au bon goût indéniable pour prénommer sa fille.

Je remercie les permanents de l'équipe et en particulier Laura Grigori, Sylvie Delaët et Gilles Fedak pour les conseils et les réponses qu'ils m'ont apportés. Les "anciens" du deuxième étage, Joffroy Beauquier et Brigitte Rozoy, pour leur expérience dont ils font profiter aux plus jeunes.

Je remercie les thésards galériens d'octobre-novembre, qui étaient dans les démarches administratives et le stress de pré-soutenance en même temps que moi, Rafael Lopez grâce à qui j'ai su qu'il fallait déposer un dossier de soutenance, Julien Clément avec qui je me suis bien amusée et j'ai pris plaisir à co-organiser le séminaire de l'équipe, et François Lesueur bien qu'il n'ait pas soutenu à Orsay.

Je remercie ceux qui ont fait que j'ai pu effectuer mes expériences dans de bonnes conditions, sur les clusters du LRI comme sur Grid'5000. L'administration de Grid'5000 est une tâche titanesque, les outils fournis sur la plate-forme en font un outil fantastique, je remercie en particulier Julien Leduc pour son travail sur Grid'5000 et sa gentillesse et sa jovialité lorsqu'il faisait partie de l'équipe puis lors de rencontres ultérieures à Bruxelles (ainsi qu'à son mariage avec Louise !). Les moyens informatiques du LRI ne seraient pas ce qu'ils sont sans la compétence et la disponibilité de l'équipe support, c'est donc pour celà que je remercie Jérôme Bernier, Laurent Darré, Raymond Douet, Olivier Lebeltel, François Zimmermann et encore une fois Vincent Neri. Enfin, je dois remercier vivement Oleg Lodygensky et Gabriel Caillat du Laboratoire de l'Accélérateur Linéaire pour m'avoir laissé utiliser cinq machines afin de constituer le petit cluster QCG.

Je remercie également les membres et visiteurs du laboratoire ICL que j'ai côtoyés pendant mes séjours à Knoxville. Emmanuel Agullo et son amour universel et sans limite pour son prochain. Julien Langou qui, avec Manu, m'a fait mettre un pied dans l'algèbre linéaire. Julie Langou, qui est assez costaud pour me porter sur son dos des courts de Golf Range jusqu'à mon appartement. Aurélie Hurault et son immense sagesse sur la condition de thésard. Don et Linda Fike, leurs cookies et leurs ailes de poulet. Aurélien Bouteiller, son calme, sa patience et sa persévérance.

Un grand merci aux personnes qui font tourner le labo et qui nous supportent au quotidien (dans tous les sens du verbe "supporter"). Les assistantes du projet Grand Large, Katia Evrat et avant elle Gina Grisvard et même celles qui ne sont pas dans Grand Large parce qu'elles sont gentilles aussi et nous aident aussi : Marie-Carol Lopes et Nicole Martin. Mention spéciale à Katia que j'ai bizutée en rentrant de Knoxville avec un genou hors service seulement quelques semaines après son arrivée dans l'équipe. Sylvie Congnard, qui officie dans la bibliothèque depuis presque aussi longtemps que j'existe et se rappelle de tous les anciens membres du labo.

Je remercie ceux dont la présence à ma soutenance m'a particulièrement touchée. La "bande de l'INT" composée de Guy Bernard, Daniel Millot et Christian Parrot, venus spécialement d'Evry et qui, pour le premier, est à l'origine de mes débuts en applications distribuées et pour les deux derniers, ont subi mes débuts en algorithmique et programmation en C (cours IF11 de Télécom INT, et les TP où on essaye de mettre un double dans un pointeur et où on ne comprend pas pourquoi ça plante).

Je remercie Sophie Voisin, amie Knoxvillienne qui m'a fait l'agréable surprise de venir assister à ma thèse lors d'un passage en France. Je remercie également les résidents de Knoxville que je n'ai pas encore cités ici, et qui ont rendu mes séjours à Knoxville si agréables que je garde un contact particulier avec le Tennessee occidental : Pauline Bouteiller, Valérie Kuentz, Céline Lemarinier, Vincent Paquit et Claudine Verney-Carron (la douairière). Delphine Gout et Olivier Gourdon et leurs conseils Iowiens.

Enfin, je remercie ceux qui m'ont soutenue durant ces trois années intensives. Mes parents qui connaissent désormais si bien la route pour aller à Roissy et son terminal 2E. Mes frères, Laurent qui a pu se libérer et venir assister à ma soutenance, et Stéphane même s'il n'a pas pu venir y assister. Mes vieilles amies Gougoute et La Marie, exhilées loin de Pontoise par leurs métiers respectifs. Romz, soutien sans faille et solide, qui m'a tenu compagnie durant de mémorables soirées de travail au labo.

Table des matières

Table des figures

Listings

Chapitre 1

Introduction

1.1 Calcul parallèle

Le calcul parallèle répond à un problème simple : pouvoir exécuter des calculs qui prendraient trop longtemps s'ils étaient exécutés sur un seul processeur sur une seule machine par un seul fil d'exécution (un seul processus, un seul thread). Certains types de calcul doivent par exemple être réalisés en un temps fixe et court. C'est le cas par exemple des simulations pour les prévisions météorologiques. Une simulation doit être effectuée du jour pour le lendemain. De plus, la fiabilité des prévisions dépend de la finesse des paramètres du calcul, notamment celle de la grille utilisée pour découper l'espace et le temps : l'augmentation de la puissance de calcul permet de résoudre un problème comportant un plus grand nombre de paramètres, et donnant donc un résultat plus précis. De même, beaucoup d'applications scientifiques nécessitent des millions d'heures de calcul : il serait alors déraisonnable d'attendre le résultat calculé par un programme séquentiel.

La méthode pour augmenter la capacité de calcul des machines a longtemps consisté à augmenter la fréquence d'horloge : depuis l'ENIAC et ses 300 opérations par secondes (environ 1000 fois plus rapide que tous les ordinateurs l'ayant précédé !), on a cherché pendant des décennies à augmenter la fréquence des micro-processeurs pour calculer plus vite. Cette course à la fréquence s'accompagne de contraintes, jusqu'alors non limitatives. La première est l'émission de chaleur : la dissipation calorifique d'un circuit électrique augmente avec sa fréquence. On fait alors face à un problème de refroidissement, qui a jusqu'à maintenant été solutionné en utilisant des systèmes d'évacuation de la chaleur de plus en plus importants (depuis longtemps bien plus gros que le micro-processeur lui-même) par ventilation puis, pour certains modèles récents, par refroidissement hydraulique. Cependant, cette approche n'est pas raisonnable indéfiniment. Par exemple le prototype de micro-processeur cadencé à 500 GHz d'IBM nécessite un refroidissement à l'hélium liquide à -269°C [57].

La seconde contrainte est liée à la taille du circuit : le courant doit pouvoir parcourir le circuit électrique d'un bout à l'autre en un temps inférieur à celui d'un cycle d'horloge. En augmentant celle-ci, on se retrouve dans l'obligation de miniaturiser toujours plus les circuits électriques. Une gravure plus fine permet alors d'augmenter le nombre de transistors sur une surface donnée afin de présenter un nombre de composants plus important et ajouter des fonctionnalités, comme augmenter la taille des caches ou le nombre d'unités de calcul. Les micro-processeurs Intel de nouvelle génération ("Nehalem") présentent des circuits dont la finesse de gravure est de 45 nanomètres actuellement, et 32 nanomètres pour les prochains. Sachant que le rayon d'un atome de sillicium est de 110 picomètres, on est à présent dans des proportions très proches de l'échelle moléculaire.

Depuis quelques années, ces deux problèmes ne peuvent plus être résolus ni contournés de la façon dont ils l'ont été jusqu'alors : la fabrication de machines plus puissantes se heurte aujourd'hui à des limites de conception et de production industrielles, et la fréquence de base des micro-proceseurs n'augmente plus aussi rapidement.

La loi (empirique) de Moore énonce le fait que le nombre de transistors présents sur les puces double tous les deux ans. La conjecture qui lui est associée a longtemps consisté à dire que la fréquence de ces puces double tous les dix-huit mois. Or, on observe depuis quelques années à un tassement de cette augmentation de la fréquence. Cependant, les circuits continuent à se complexifier en offrant d'autres façons d'augmenter leur puissance de calcul notamment en possédant plusieurs unités de calcul (architecture multi-cœurs).

L'approche suivie est alors de paralléliser les calculs. La méthode pour obtenir de meilleures performances

théoriques est alors d'augmenter le parallélisme et donc le nombre d'unités calculant simultanément [1]. Cette approche n'est pas nouvelle en elle-même, puisque les toutes premières machines étaient déjà massivement parallèles. Les processeurs vectoriels permettent d'appliquer une instruction sur un vecteur de données et ainsi de mettre en place un parallélisme de données.

Les super-calculateurs sont aujourd'hui tous des machines parallèles. Si, jusqu'au début des années 80, le Top 500 [2] recensait quelques machines constituées d'un seul processeur (généralement vectoriel), celles-ci ont été remplacées par des machines multi-processeurs (SMP) puis par des machines massivement parallèles, jusqu'à l'avènement des grappes de calcul depuis la seconde moitié des années 90.

À plus petite échelle, les micro-processeurs ne sont pas en reste : après l'introduction des processeurs superscalaires et l'augmentation progressive de la profondeur du pipeline a été introduite la technique d'*hyper-threading*, permettant à un seul processeur de traiter plusieurs threads en même temps. Puis, s'inspirant des architectures multi-processeurs, les processeurs multi-cœurs ont permis d'avoir, sur une seule puce, plusieurs unités de calcul.

Enfin, on adjoint souvent aujourd'hui des *accélérateurs* au micro-processeur central, en exploitant la puissance de calcul des cartes graphiques ou de processeurs comme le Cell.

Ces architectures où la puissance de calcul est distribuée demandent des techniques de programmation adaptées pour être exploitées pleinement. Les calculs à effectuer doivent être *parallèles*, c'est-à-dire qu'ils doivent pouvoir être constitués de plusieurs fils d'exécution (threads ou processus, selon l'architecture [3]) s'exécutant en parallèle sur les ressources de calcul mises à leur disposition.

Les machines parallèles sont souvent classifiées selon la taxonomie de Flynn [77] qui les range en quatre catégories :

- *Single Instruction, Single Data (SISD)* : il s'agit des machines traditionnelles à processeur scalaire unique, ne traitant qu'une seule instruction à la fois sur une seule donnée ;
- *Single Instruction, Multiple Data (SIMD)* : plusieurs données sont traitées en même temps, mais la séquence d'instruction exécutées sur elles est la même. On peut citer les processeurs vectoriels ou les unités de calcul des cartes graphiques dans cette catégorie ;
- *Multiple Instruction, Single Data (MISD)* : aucune machine parallèle ne rentre dans cette catégorie qui traite une seule donnée par plusieurs séquences d'instructions différentes. Cependant on peut citer des systèmes critiques utilisant différents calculateurs sur un même flux de données afin de vérifier la justesse du résultat, comme les systèmes utilisés dans l'avionique et l'aéronautique ;
- *Multiple Instruction, Multiple Data (MIMD)* : plusieurs données sont traitées en même temps par des séquences d'instructions différentes. Il s'agit du cas général d'une machine parallèle à mémoire partagée ou distribuée.

Les applications de calcul parallèle sont donc déterminés par la façon de traiter deux entités : les instructions et les données. Pour cela, on dispose de techniques de programmation des applications permettant de définir la séquence d'instruction à exécuter sur des données, et de mettre à disposition ces données.

La programmation de machines parallèles nécessite la prise en compte de ce parallélisme, permise par l'utilisation d'un environnement de programmation parallèle. Celui-ci permet de mettre en relation les différents composants de l'application parallèle et d'organiser l'application.

1.2 L'environnement de programmation parallèle

Dans [39] un environnement de programmation parallèle est décomposé en trois éléments :

- un système de lancement, qui déploie et lance l'exécution des processus sur les ressources de calcul disponibles ;
- un environnement d'exécution, qui met les processus en relation les uns avec les autres, transmet les entrées-sorties et les signaux et assure la surveillance et la finalisation de l'application ;
- une bibliothèque de communication, qui effectue des mouvements de données entre les processus.

1. On parle ici de performances théoriques car il est ensuite nécessaire de pouvoir exploiter ce parallélisme au niveau logiciel
2. Top500 : `http://www.top500.org`
3. Par abus de langage et dans un soucis de généralité par rapport au modèle de parallélisme, on parlera de *processus* dans la suite pour désigner la plupart du temps un processus ou un thread.

Le système de lancement peut être intégré à l'environnement d'exécution (comme c'est le cas quand on utilise un lanceur d'applications distantes de type SSH) ou cette tâche peut être déléguée à une application tiers (système de lancement souvent intégré à un système de réservations), mais en s'interfaçant avec l'environnement d'exécution.

1.2.1 Le support exécutif

Les applications parallèles ont besoin pour s'exécuter d'être supportées par un intergiciel qui sert d'intermédiaire entre l'application et les machines sur lesquelles elle s'exécute. Le rôle de ce support exécutif est de lui permettre de tirer parti du matériel mis à sa disposition, si possible de la façon la plus efficace possible [131].

Le support exécutif assure la portabilité de l'application d'une machine à une autre et d'un système (grappe, machine multi-processeurs...) à un autre. Par exemple, dans le contexte de communications suivant la norme MPI les processus sont *nommés* par un rang. Les rangs des processus sont attribués de façon déterministe entre 0 et $(N-1)$, si N désigne le nombre de processus de l'application. Cette abstraction assure une partie de la portabilité des communications entre les processus : au niveau du matériel les processus communiquent les uns avec les autres grâce à des informations de connexion, c'est-à-dire, pour des communications TCP, un couple *(adresse IP ; port)*. D'une exécution à une autre le port utilisé par une machine peut être différent. De plus si les machines utilisées pour l'exécution ne sont pas exactement les mêmes, l'adresse IP ne sera pas la même.

Enfin, le support exécutif assure une portabilité matérielle. En restant dans l'exemple des communications entre processus d'une application parallèle, les types de réseaux disponibles peuvent changer d'une exécution à une autre en changeant de machines. Dans certains cas simples le système d'exploitation peut remplir une partie des rôles du support exécutif : il peut souvent par exemple gérer les communications par passage de messages sur des sockets ou en utilisant explicitement de la mémoire partagée. Ce n'est pas le cas de tous les types de réseaux et certains nécessitent des pilotes fournis par des libraries externes pour être utilisés. De même, l'application peut être exécutée sur une grappe qui utilise un certain type de réseaux à hautes performances, puis sur une autre grappe disposant de deux interfaces réseaux à hautes performances pour un autre type de réseaux. Le type de réseau d'interconnexion est caché à l'application par l'abstraction des primitives de communications fournie par le support d'exécution.

Ainsi, le support exécutif offre une abstraction des moyens de communications mis à disposition de l'application en lui fournissant des routines de communications de haut niveau et en prenant en charge la gestion du bas niveau. L'application peut alors se concentrer sur les communications en elles-mêmes et non pas sur le moyen de les effectuer.

1.2.2 Modèle pour la programmation parallèle

On doit alors effectuer une distinction entre les différents modèles de programmation d'applications parallèles. En effet, le support apporté par l'environnement d'exécution diffère selon le modèle dans lequel on se place.

On distingue deux grands types de machines parallèles : celles à mémoire partagée et celles à mémoire distribuée. Dans le cas des machines à mémoire partagée, tous les fils d'exécution ont accès à la mémoire de la machine. Une seule instance du système d'exploitation est exécutée sur la machine et les processus peuvent communiquer en utilisant des mécanismes de communications inter-processus comme des segments de mémoire partagée, des verrous et des sémaphores.

Dans le cas des machines à mémoire distribuée, les processus n'ont pas accès à toute la mémoire mise à la disposition de l'application ; seuls quelques processus ont accès à une zone donnée de mémoire. Il est alors souvent nécessaire de procéder à des déplacements de données pour que d'autres processus aient accès à des données situées dans une zone de mémoire à laquelle ils n'ont pas accès. Les processus doivent alors être en mesure d'accéder à un réseau permettant de les relier les uns aux autres, et d'effectuer des communications sur ce réseau.

Ces communications peuvent être *explicites*, dans le cas où il est écrit dans le programme qu'il faut effectuer une communication avec un ou plusieurs processus, ou *implicites*, dans le cas par exemple des langages parallèles à espace d'adressage global, où la localité des données est gérée par le compilateur ou par un système d'exploitation distribué à image unique. Dans le premier cas, l'environnement d'exécution doit permettre d'effectuer ces communications en proposant une correspondance entre un adressage interne à l'application (par exemple, en numérotant les processus) et des informations de connexion qui permettent d'effectuer les communications mais

ne sont pas montrées explicitement à l'application. Dans le second cas, l'environnement d'exécution est chargé d'effectuer une traduction d'adresse de l'espace global virtuel vers une machine et son espace d'adressage local.

Cette thèse s'intéresse aux environnements d'exécutions pour applications parallèles dans un modèle à mémoire distribuée et communiquant par passage de messages explicites, c'est-à-dire dans le premier cas présenté au paragraphe précédent. Les applications sont effectuées dans l'instance de ce modèle que constitue la norme MPI [79], qui standardise une interface de programmation d'applications parallèles conformes au modèle SPMD en définissant des fonctions de communications point-à-point et collectives.

1.2.3 L'environnement d'exécution

Le rôle de l'environnement d'exécution est alors de permettre l'exécution de l'application. Dans le contexte présent, il doit pour cela lui rendre des services :

- déploiement, lancement et finalisation de l'application comme un tout, en prenant en charge le traitement individuel des processus ;
- mise en relation des processus de l'application pour qu'ils communiquent les uns avec les autres ;
- surveillance : détection de faute, comportement à tenir en cas de mort d'un processus. La surveillance peut être étendue à une surveillance plus fine de l'état des machines, comme la surveillance des données d'état matériel telles que la charge des processeurs ou sa température. L'environnement d'exécution choisit alors de lancer ou pas des processus ou de les migrer en cours d'exécution sur une machine plus saine.

Dans le contexte des langages parallèles à espace d'adressage partitionné, qui présente quelques différences mais aussi des similitudes avec le contexte étudié ici, un inventaire des fonctionnalités qu'un environnement d'exécution doit fournir à une application parallèle a été proposé dans [84]. Il est énoncé le fait qu'un environnement d'exécution doit fournir à l'application un support portable d'une machine à une autre et capable de passer à l'échelle. Il doit permettre aux processus de pouvoir communiquer entre eux et, dans le cas des langages parallèles, est également chargé des mouvements de données. Dans le cas des modèles de programmation par passage explicite de messages comme MPI, le rôle de l'environnement d'exécution se limite au fait de rendre *possible* cette communication ; les communications sont effectuées de manière explicite par la bibliothèque de communications.

L'environnement d'exécution est généralement composé d'un ensemble de processus (démons) exécutés sur les machines utilisées dans un calcul. Chaque processus de l'application est géré par un démon de l'environnement d'exécution, avec lequel elle peut communiquer. Le démon est généralement exécuté sur la même machine que les processus qu'il gère, à quelques exceptions près [4]. Il peut être exécuté de façon persistante sur l'ensemble des ressources disponibles et supporter plusieurs applications à la fois, ou être dédié à une application en particulier.

Dans ce découpage, on constate que le support exécutif défini dans la section 1.2.1 englobe l'environnement d'exécution et les couches basses de la bibliothèque de communications. Il inclut l'environnement d'exécution dans le sens où il assure l'exécution des processus de l'application sur les ressources matérielles mis à sa disposition et les couches basses de la bibliothèque de communication en fournissant une abstraction des interfaces réseaux disponibles et des méthodes de communications (comme par exemple les communications collectives, qu'elles soient disponibles ou non sur le type de réseau physique utilisé).

Les environnements d'exécutions destinés aux implémentations de la norme MPI fournissent des fonctionnalités strictement liées au cycle de vie de l'application. Par exemple, les communications de l'application sont prises en charge par la bibliothèque de communications : cet ensemble de fonctionnalités fait partie du support d'exécution mais ne rentre pas dans les rôles rendus par l'environnement d'exécution. De plus, on suppose que l'ordonnancement est effectué par un service externe de réservation et d'attribution des ressources. Enfin, les services de gestion du cycle de la vie de l'application tels que le lancement, la surveillance et éventuellement la tolérance aux pannes n'est pas prise en charge par le support d'exécution puisqu'il ne s'agit pas d'assurer la portabilité de l'application et l'exploitation des ressources matérielles qu'elle peut utiliser.

L'environnement d'exécution a besoin pour fournir ses services d'une infrastructure de communications sur laquelle circulent des messages internes à son fonctionnement, dits hors-bande. Cette infrastructure doit former un graphe couvrant sur l'ensemble des processus de l'environnement d'exécution, afin de pouvoir joindre tous

4. Pour les supercalculateurs comme BlueGene/L qui ne supportent qu'un seul processus à la fois et où la notion de nœud de calcul ne peut pas être définie par "une instance du système d'exploitation", l'architecture de l'environnement d'exécution sera présentée ultérieurement.

les processus de l'application. Certains environnements d'exécution sont alors spécialisés dans la mise en place de cette infrastructure et les communications effectuées dessus : par exemple, MRNet [10, 147] est un réseau de couches fournissant des outils de communications pour des systèmes parallèles et distribués. Les communications se basent sur les deux primitives de diffusion et de réductions, permettant de transférer et d'agréger des données entre des nœuds selon une topologie capable de passer à l'échelle. La topologie utilisée est un arbre, choisi pour ses qualités de scalabilité. Il est principalement utilisé pour déployer des outils comme des processus de surveillance de l'état d'une application et/ou des ressources ou de débuggage [118].

1.3 Évolution des environnements d'exécution

Il est possible de recenser les services que l'environnement d'exécution fournit à une application parallèle en général. On peut alors fixer une interface bien définie de services rendus à l'application. Ainsi, un environnement d'exécution pourra être utilisé par plusieurs types d'applications.

Un effort dans ce sens a été entrepris au début des années 1990 pour les langages parallèles à espace d'adressage partitionné (PGAS) en essayant de définir le support que doit fournir l'environnement des exécutions [84]. Les fonctionnalités recensées par ce consortium de chercheurs dans le domaine des langages PGAS sont classées en trois catégories : support du parallélisme de tâche, support de parallélisme de données et possibilité d'utiliser l'infrastructure de l'environnement d'exécution pour des outils de mesure de performances et de débuggage.

Outre l'expression des besoins d'une application parallèle vis-à-vis de l'environnement d'exécution qui la supporte, ce travail est né de la volonté d'unir les efforts de différents projets sur un composant commun de l'environnement parallèle, au lieu de créer autant d'environnements d'exécution qu'il existe d'implémentation de techniques de programmation parallèles, et de refaire ce qui a déjà été fait auparavant.

Une des exigences pour un environnement d'exécution commun est la possibilité de supporter différents langages de programmation. On peut étendre cette condition en proposant de supporter différents modèles de programmation parallèle.

C'est en ce sens que quelques années après est apparu HARNESS [75, 73], un environnement d'exécution générique. Cette généricité s'exprime par le fait d'exporter une interface permettant d'accéder aux services rendus par l'environnement d'exécution, et de pouvoir utiliser cette interface dans une application parallèle, quel qu'elle soit, sans qu'aucune hypothèse sur le modèle de programmation cible n'ait été faite lors de la conception de HARNESS. Ainsi, il est utilisé par PVM, une machine virtuelle parallèle [154], et par FT-MPI, une bibliothèque de communication par passage de messages [74]. Les débuts du développement de MPD [39] avaient pour ambition d'en faire un support pour le lancement et l'exécution de programmes parallèles en général, sans se centrer sur la bibliothèque MPI pour laquelle il a originellement été créé, MPICH2 [37].

Les langages PGAS ont également unifié leur environnement d'exécution avec GASNet [19], qui est utilisé par Berkeley UPC [106], Titanium [102] et l'implémentation du Laboratoire National de Los Alamos de CoArray Fortran [55]. Une tentative d'utilisation de la bibliothèque MPI pour programmer un environnement d'exécution pour UPC a été effectuée [20] afin d'essayer de rapprocher les couches basses des langages PGAS de celles de MPI : ainsi, la bibliothèque MPI assurerait les communications nécessaires aux accès mémoire distants pour supporter l'exécution des programmes PGAS, et l'environnement d'exécution de la bibliothèque MPI utilisée fournirait les fonctionnalités communes à toutes les applications parallèles, par passage de messages ou à espace d'adressage partitionné. Cependant, le modèle de communications MPI n'est pas adapté aux besoins des communications des langages PGAS et ne permet pas d'obtenir de bonnes performances.

Les expériences de HARNESS et MPD ont montré les limites de cette approche. Par exemple, MPD avait pour but d'être simple et ce pour plusieurs raisons : pour garantir un maximum de fiabilité du code, de simplicité des mécanismes utilisés, pour être le plus générique possible... Cependant, cet objectif a contraint à faire des concessions sur d'autres aspects. Par exemple, la topologie utilisée par l'infrastructure de communications est un anneau. Le routage dans un anneau est très simple et il est facile de le rendre résilient (de le faire cicatriser après la mort d'un processus de l'anneau). Cependant, cette topologie ne passe pas à l'échelle et est un sérieux frein à la performance de MPD. HARNESS s'est lui montré peu extensible, et finalement peu flexible sur les services rendus aux applications qu'il supporte : il offre peu d'abstraction sur son modèle de fonctionnement, et l'application doit en tenir compte. Par exemple, les processus applicatifs sont synchronisés lorsqu'un processus HARNESS meurt. Ce comportement n'est pas forcément désiré par toutes les applications parallèles.

La génération suivante s'est donc rapprochée d'un modèle où chaque implémentation d'un modèle de programmation a son propre environnement d'exécution, afin de pouvoir le contrôler et interagir avec lui plus finement et de ne pas payer le coût de la généricité. MPD s'est donc spécialisé pour MPICH2. Lors de la création du projet OpenMPI [85], il a été décidé de créer OpenRTE [46], un nouvel environnement d'exécution spécifique à cette implémentation de MPI.

Ces projets séparés ont permis de mener des projets distincts sur les nouvelles problématiques émergentes entre les différentes équipes de développement de bibliothèques parallèles. Des approches différentes entre les équipes ont permis d'explorer différentes façons de faire face à ces nouveaux défis qui ont causé la fin des environnements d'exécutions génériques. Cette dissémination des efforts de recherche a également eu l'inconvénient de répliquer certains efforts de recherche.

La tendance est aujourd'hui à nouveau au rassemblement des forces pour un environnement d'exécution de nouvelle génération unifié. STCI [35] se veut une infrastructure générique pour le support d'applications parallèles en général. Il s'agit d'un projet commun entre les équipes de plusieurs environnements de programmation (OpenMPI, MPICH2, PVM). Les services présentés à l'application ont été définis en recensant ce qui est nécessaire à une application parallèle en général, sans chercher à s'adapter à un type d'applications en particulier. Ainsi, par exemple, une couche de communications sera disponible pour la plupart des réseaux utilisés de manière courante dans les grappes et les supercalculateurs. De cette manière, STCI pourra fournir des communications à hautes performances sur son infrastructure de communications, et non pas, comme c'est le cas actuellement, uniquement des connexions de signalisations qui n'essayent pas d'utiliser les réseaux rapides disponibles afin de les laisser entièrement disponibles pour l'application.

Cette unification permet également de rassembler les expériences acquises lors de la période où chaque environnement de communication avait son propre environnement d'exécution, et de réunir les meilleures solutions proposées par les uns et les autres.

On constate donc que la nature des relations entre les environnement d'exécution et les bibliothèques de programmation est faite de cycles où les efforts tendent tantôt à s'unifier autour d'un projet commun à plusieurs environnements de communications, tantôt, pour faire face à de nouveaux défis, à se disperser pour tenter d'y répondre de la façon spéccifique aux besoins des applications ciblées.

1.4 Axes de recherches

Si les bibliothèques de communications ont fait l'objet de nombreux efforts d'optimisation tant en performances pures que du point de vue de leur passage à l'échelle, moins d'attention a été portée à la scalabilité des environnements d'exécution. L'attention a jusque là été portée sur des fonctionnalités prises une par une. Par exemple, l'infrastructure de communications pour MRNet, orientée résolument vers la grande échelle mais qui ne fournit pas de mécanisme d'auto-déploiement.

Cette thèse s'intéresse aux problématiques liées à la grande échelle sur l'environnement d'exécutions, et sur le support que celui-ci apporte à l'application. Pour cela, elle suit trois axes présentés dans ce qui suit.

1.4.1 Passage à l'échelle

Le premier axe de recherches concerne le passage à l'échelle de l'environnement d'exécution lui-même, de ses fonctionnalités internes et donc des services qu'il rend à l'application.

Comme présenté ci-avant, l'augmentation de la puissance de calcul disponible pour une application passe par la distribution des ressources. L'environnement d'exécution doit alors supporter des processus exécutés sur un grand nombre de ressources de calcul, qui peuvent souvent exécuter elles-mêmes plusieurs processus. Alors que les supercalculateurs les plus puissants à l'heure actuelle comptent plusieurs dizaines voire centaines de milliers de cœurs, ces cœurs sont contrôlés par l'environnement d'exécution par quelques centaines à quelques milliers de démons. Il est alors nécessaire de pouvoir supporter de manière efficace l'application à une telle échelle, en ayant un environnement d'exécution capable de passer à l'échelle et de contrôler plusieurs processus par démon.

Des retours d'expérience d'exécutions d'applications à grande échelle montrent un temps utilisé par l'environnement d'exécution qui n'est plus négligeable. Par exemple, MPICH2 [37] a besoin d'une demi-heure pour déployer une application sur 1 024 processus avec MPD [39] sur la grille de calcul Grid'5000 [41], et nécessite des réglages adaptés des paramètres TCP des machines utilisées afin de supporter les tempêtes de connexions

réseaux au-delà d'une centaine de nœuds. Un lancement sur 4 096 nœuds de la machine Thunderbird du Laboratoire National de Los Alamos en 2006 durait 25 à 30 minutes avec OpenMPI [85] en utilisant ORTE [46] et plus de trois heures avec MPICH2 en utilisant MPD. À grande échelle, le temps de lancement devient alors significatif dans le temps d'exécution d'une application et a un impact non négligeable sur la rentabilité de la machine.

Outre le temps de lancement, les communications au sein de l'environnement d'exécution peuvent être un facteur critique de performances. Le choix est souvent fait de ne pas les faire passer par les réseaux rapides, s'il y en a, afin de ne pas perturber les communications de l'application. Elles passent donc par l'infrastructure de communications construite par l'environnement d'exécution, qui doit être capable de supporter une charge lourde. L'application Ray2mesh [100] est relativement verbeuse ; son utilisation sur une grille de 768 processeurs avec LAM-MPI [38] a nécessité pour un calcul s'exécutant en huit minutes une attente de une heure dix pour obtenir l'intégralité des messages affichés sur la sortie standard.

Le chapitre 3 présente l'étude de mécanismes qui peuvent améliorer la scalabilité d'un environnement d'exécution, en proposant une construction de l'infrastructure de communications efficace et la rendant capable de passer à l'échelle.

1.4.2 Tolérance aux pannes

Des études statiques [145] montrent que lorsque le nombre de composants dans un système augmente, la probabilité qu'un de ces composants subisse une panne augmente. Le temps moyen avant défaillance dans les systèmes les plus grands actuellement (plusieurs dizaines ou centaines de milliers de cœurs) est faible (quelques heures seulement) et les pannes sont devenues un obstacle important à l'efficacité des applications à grande échelle.

Il est alors indispensable de pouvoir tolérer ces pannes. Le rôle de l'environnement d'exécution est alors étendu afin de supporter cette tolérance aux pannes. Un protocole de tolérance aux pannes est mis en place au niveau de l'application pour restaurer son état et permettre au calcul de se poursuivre ; l'environnement d'exécution doit fournir à ce protocole les éléments de support nécessaires à le rendre possible.

Le fait de rendre possible la tolérance aux pannes a une influence sur le support du cycle de vie de l'application. Dans ce cas l'environnement d'exécution doit pouvoir, selon le protocole utilisé, redémarrer l'application à partir d'un état donné et non pas en relançant l'application entièrement ou survivre à une panne et continuer à remplir son rôle auprès des autres processus tout en permettant l'exécution du protocole de rétablissement de l'état des processus victimes de la panne. Dans le second cas il doit uniquement terminer l'exécution des processus survivants.

De plus, des fonctionnalités doivent être ajoutées à l'environnement d'exécution afin de permettre le déroulement de ces protocoles de tolérance aux pannes durant l'exécution de l'application, la plupart du temps en permettant l'enregistrement de données sur l'exécution de l'application.

Le chapitre 4 présente une étude des mécanismes de tolérance aux pannes pour les applications parallèles, et propose un protocole permettant à l'environnement d'exécution de supporter les mécanismes de tolérance aux pannes de façon scalable.

1.4.3 Cas des grilles de calcul

Les grilles de calcul sont un cas particulier de systèmes à grande échelle : elles sont constituées en mettant en commun plusieurs grappes ou supercalculateurs. Des problématiques liées à cette mise en commun de ressources apparaissent alors.

Le projet QosCosGrid(*Quasi-Opportunistic Supercomputing for Complex Systems in Grid environemnts*) a pour but de permettre l'exécution d'applications parallèles sur des grilles de calcul. Ce projet a été financé par l'Union Européenne et rassemble onze institutions partenaires en Europe, Irsaël et Australie [5]. Les applications cibles sont des systèmes complexes, qui présentent un certain nombre de caractéristiques communes. Toutes ces applications présentent des besoins importants en capacités de calcul et de stockage, c'est pourquoi le choix d'un matériel por exécuter de telles applications s'est porté vers les grilles de calcul.

5. Le partenaire Australien a été financé par l'Australian Science Foundation.

Celles-ci posent des problèmes de connectivité entre les processus situés sur différentes grappes : en effet, des raisons de sécurité imposent d'interdire les accès extérieurs aux grappes en utilisant des équipements de sécurité tels que les pare-feux et la traduction d'adresses. Or les applications exécutées sur ces grilles de calcul nécessitent la possibilité de communiquer entre processus situés sur différentes grappes, et donc entre les grappes.

L'environnement d'exécution permet de prendre en charge ces contraintes dans le cadre de son rôle dans la mise en relation des processus entre eux. Il permet notamment de mettre en place des techniques avancées de communications entre les processus de l'application.

Les chapitre 5 présente QCG-OMPI, un environnement de programmation et d'exécution d'applications parallèles communiquant par passage de messages conçu spécifiquement pour les grilles et répondant à des problématiques de connectivité qui lui sont spécifiques.

On s'intéresse ensuite à la programmation de tels systèmes. En effet, les performances des liens de communications sur de tels systèmes diffèrent de plusieurs ordres de grandeur selon si deux processus communiquant l'un avec l'autre sont situés sur la même machine, sur la même grappe ou dans des grappes différentes. On constate généralement trois ordres de grandeur entre deux niveaux de hiérarchie consécutifs. Il est donc nécessaire d'adapter les schémas de communication et, par conséquent, de calcul des applications destinées à être exécutées sur de tels systèmes afin de prendre en compte cette hiérarchie.

Or, on se heurte ici à un problème de portabilité : l'environnement matériel pour lequel l'application a été conçue ne sera pas forcément retrouvé d'une exécution à une autre. Il est alors possible de mettre à contribution l'environnement d'exécution de l'application avec le méta-ordonnanceur de grille en lui spécifiant la topologie nécessaire à l'application pour s'exécuter efficacement. Cette assurance de retrouver la même topologie physique ou une topologie équivalente lors de chaque exécution rend l'écriture d'applications parallèles pour systèmes hiérarchiques beaucoup plus simple, puisque l'adaptation de l'application à la topologie est effectuée de façon statique et non plus dynamique comme il est nécessaire de le faire lorsque la topologie physique n'est pas connue à l'avance et est découverte lors de l'exécution. Lors de l'exécution de l'application, l'application peut s'organiser autour de cette topologie en retrouvant sa description grâce à l'environnement d'exécution.

Le chapitre 6 présente comment l'environnement de programmation peut être étendu et bénéficier de fonctionnalités de l'environnement d'exécution présenté au chapitre précédent pour programmer des applications efficaces sur les grilles.

1.5 Plan de ce document

Le présent document est organisé comme suit. L'environnement logiciel dans lequel ont été intégrés les mécanismes étudiés dans cette thèse, ainsi que la plate-forme expérimentale qui a été utilisée pour les évaluations de performances, sont présentés brièvement dans le chapitre 2 ; cet environnement est constitué de composants logiciels, dans lesquels ont été mis en place les principes et les mécanismes exposés dans le corps de cette thèse, ainsi qu'une plate-forme expérimentale sur laquelle ont été effectuées les expériences d'évaluation. Les chapitres techniques constituant le corps de cette thèse sont les chapitres 3 à 6. Le chapitre 3 présente des travaux portant sur le passage à l'échelle des fonctionnalités de l'environnement d'exécution. Le chapitre 4 détaille le support qu'il doit apporter pour la tolérance aux pannes dans les applications parallèles. Enfin, les deux chapitres 5 et 6 présentent des travaux effectués dans un environnement de grilles de calcul, présentant respectivement des mécanismes permettant l'exécution d'applications parallèles sur des grilles et comment dans ce contexte les services qu'il rend à l'application peuvent être étendus pour améliorer les performances. Par ailleurs, chaque chapitre comporte une étude comparative d'environnements d'exécution présentant des fonctionnalités proches ou liées au sujet étudié dans le chapitre.

Chapitre 2

Environnement d'évaluation

Les mécanismes et les algorithmes implémentés pour évaluer les solutions proposées ont été intégrés dans des implémentations existantes de la norme MPI : MPICH2 et OpenMPI. Les évaluations de performances ont été effectuées sur la plate-forme expérimentale Grid'5000, qui est une grille de calcul dédiée à la recherche en informatique.

Ce chapitre présente dans un premier temps les bibliothèques MPI utilisées. Les mécanismes présentés et étudiés dans cette thèse ont conduit à des modifications plus importantes dans OpenMPI que dans MPICH ; OpenMPI fait donc l'objet d'une présentation plus détaillée et présente également des contributions que j'ai apportées à son fonctionnement. La seconde partie de ce chapitre présente la plate-forme expérimentale qui a servi à effectuer les évaluations de performances.

2.1 Implémentations

2.1.1 MPICH2

MPICH2 est le successeur de MPICH [99], développé au Laboratoire National d'Argonne. Il s'agit d'une nouvelle implémentation, implémentant la norme MPI 2.0, remaniée pour permettre notamment le support des opérations de gestion de processus dynamiques.

2.1.1.1 Architecture

L'architecture de MPICH2 est représentée figure 2.1(a). Au plus haut niveau on trouve ADI3, qui interface les fonctions exhibées par la bibliothèque (les routines MPI) avec les fonctions internes de communication de MPICH2.

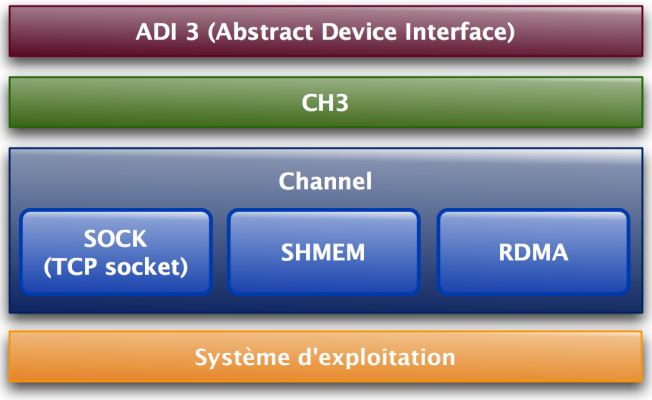
Ces fonctions de communications sont implémentées dans CH3 (Chameleon) et font appel aux fonctions implémentant les communications au plus bas niveau, selon les types de réseaux utilisés. Les communications réseau sont implémentées dans le canal (ou Channel), avec un canal par type de réseaux. Le canal pour utiliser un segment de mémoire partagée entre deux processus, SHMEM, est toujours inclus en utilisable en même temps que tout autre canal disponible.

Le choix du pilote réseau se fait à la compilation de MPICH2. Il n'est pas possible d'utiliser un nouveau type de réseaux sans recompiler la bibliothèque.

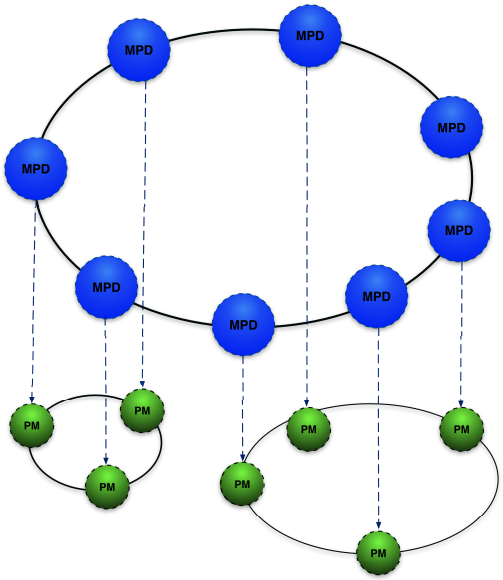
Chaque couche est appelée par la couche supérieure par des crochets (*hooks*). L'implémentation d'un protocole de communications consiste à implémenter les crochets du canal.

2.1.1.2 Environnement d'exécution

L'environnement d'exécution de MPICH2 est appelé MPD. Il s'agit d'un ensemble de démons lancés par une commande extérieure (`mpdboot`) et exécutés de manière persistente sur les nœuds. Il n'est pas finalisé à la fin de l'exécution en même temps que l'application mais peut être utilisé par plusieurs applications simultanées ou successives. Il n'est pas rattaché à une application MPI en particulier.



(a) Architecture de MPICH2.



(b) Environnement d'exécution de MPICH2.

Lorsqu'une application MPI est lancée, les démons MPD sont clonés et exécutent les gestionnaires de processus (PM), qui forment l'instance de l'environnement d'exécution dédiée à cette application en particulier. Les PM lancent l'application et assurent les services rendus par l'environnement d'exécution à l'application. Ce fonctionnement est illustré par la figure 2.1(b). On y voit les MPD interconnectés en anneau et les PM lancés pour deux applications MPI. Les PM sont également connectés au `mpiexec`, non représenté sur cette figure.

Les démons MPD sont connectés selon une topologie en anneau [89]. Ils sont volontairement simples et ne fournissent qu'un nombre limité de services : en effet, ils doivent pouvoir être exécutés de manière persistante sur les machines, éventuellement par le super-utilisateur. Pour des raisons de sécurité, il doivent réaliser un nombre limité et bien connu d'opérations simples. La topologie en anneau est un choix qui a été fait en ce sens : elle est simple, mais ne passe pas à l'échelle au-delà de quelques dizaines ou centaines de machines.

Les services rendus par les MPD ont été définis dans la spécification BNR. Celle-ci définit les opérations basiques nécessitées par le cycle de vie d'un processus MPI. BNR est implémentée par l'interface PMI (*Process Management Interface*), qui permet de réaliser ces opérations sur un groupe de processus correspondant à une application MPI parmi celles qui sont gérées par MPD.

MPD est tolérant aux pannes dans le sens où il est capable de reformer sa topologie d'anneau en réduisant sa taille lorsqu'un démon meurt : les démons de part et d'autre du démon mort se connectent l'un à l'autre et rétablissent l'anneau, même en cas de fautes simultanés de démons consécutifs.

2.1.2 OpenMPI

OpenMPI [85] est une implémentation de nouvelle génération développée par un consortium formé de laboratoires universitaires, d'instituts de recherche et d'industriels. Cette implémentation est née de la mise en commun de l'expérience des équipes de développement de LAM/MPI, LA-MPI, PACX-MPI et FT-MPI.

2.1.2.1 Architecture

OpenMPI est composé de trois sections :
- une couche de bas niveau, assurant la portabilité (OPAL)
- l'environnement d'exécution et la couche d'interaction avec l'environnement d'exécution (ORTE)
- l'implémentation des routines MPI (OMPI)

Il ne s'agit pas à proprement parler de *couches*, comme l'illustre le schéma 2.1 où sont représentées les interactions entre les sections.

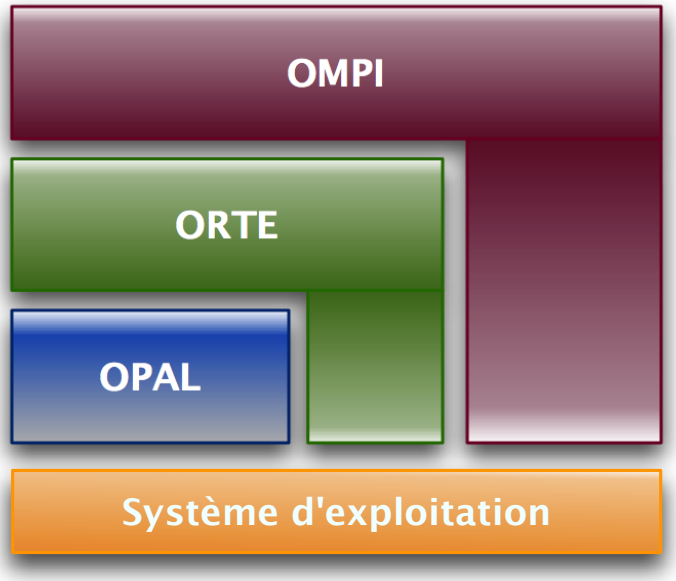


FIGURE *2.1 – Vue générale de l'architecture d'OpenMPI.*

L'architecture d'OpenMPI est modulaire et repose sur MCA, une architecture de composants. Chaque fonctionnalité interne est assurée par un *framework*, qui fournit une API implémentée dans des *composants*. Un composant implémente une façon de remplir la tâche du framework, par exemple en utilisant un certain type de réseaux. Pour certains frameworks, plusieurs instances des composants peuvent être utilisées en même temps. Au moment de l'exécution, une instance d'un framework chargée par le système et utilisée par OpenMPI est appelée un *module*.

Chaque framework remplissant une tâche particulière, ils ne dépendent pas forcément les uns des autres comme dans une architecture en couches. Par exemple, il n'y a pas de relation de ce type entre le framework chargé du déploiement des démons et le framework chargé des communications hors-bande.

2.1.2.2 Environnement d'exécution

L'environnement d'exécution d'OpenMPI est un ensemble de démons lancés par la commande de démarrage de l'application (mpiexec, mpirun ou orterun). Un démon est exécuté par nœud, et les processus MPI disposent d'une couche environnement d'exécution leur permettant de s'interfacer avec ce démon.

La figure 2.2 présente les frameworks d'ORTE et leurs interactions. On peut les classer en trois catégories :

- les frameworks gérant le cycle de vie d'ORTE et de l'application, c'est-à-dire son lancement et la gestion des erreurs ;
- les frameworks composant l'infrastructure de communications hors-bande ;
- les outils, intervenant sur tout l'environnement d'exécution lui-même.

Cycle de vie d'ORTE et d'une application

Resource Allocation System (RAS) Le RAS est chargé d'obtenir les ressources sur lesquelles l'application va être exécutée, en lisant un fichier de machines fourni par l'utilisateur ou par un système de réservation de ressources. Les composants permettent de supporter différentes méthodes d'obtention des ressources, selon le système de réservation utilisé.

Resource Mapping System (RMAPS) Le RMAPS est chargé d'établir une correspondance entre les processus de l'application MPI, les démons de l'environnement d'exécution et les machines disponible obtenues par le RAS. Les composants permettent d'utiliser différentes politiques d'ordonnancement des processus sur les ressources disponibles, par exemple en round-robin.

Process Lifecycle Management (PML) Le PLM est chargé de déployer l'environnement d'exécution selon la carte établie par le RMAPS. Il lance les démons sur les ressources distantes, et s'assure qu'ils rejoignent tous l'infrastructure de communications. Une fois tous les démons prêts, il propage la commande qui lance l'application sur tous les démons.

OpenRTE Daemon's Local Launch Subsystem (ODLS) L'ODLS est chargé, localement, de lancer les processus de l'application qui seront rattachés au démon. Pour cela, il utilise la carte établie par RMAPS et clone le processus démon orted pour lancer les processus MPI.

Infrastructure de communications

ROUTED L'établissement des tables de routages de l'infrastructure de communications est effectué par le framework ROUTED. Il calcule les identifiants des démons par lesquels les messages seront acheminés pour arriver à destination. Les tables de routage contiennent au niveau de ROUTED uniquement les identifiants des démons, et non pas les adresses vers lesquelles les messages seront envoyés. Chaque composant correspond à un algorithme de routage.

Out-Of-Band (OOB) Les communications de bas niveau sont implémentées dans OOB. Afin de ne pas perturber les communications de l'application si elles ont lieu sur un réseau rapide, seul le composant implémentant ces communications sur réseaux TCP est implémenté.

Runtime Messaging Layer (RML) Les communications utilisées par l'environnement d'exécution sont implémentées dans le RML, qui appelle OOB. RML suit le modèle d'*active message*. Il s'agit de communications de plus haut niveau que celles de OOB, permettant par exemple de poster des requêtes non-bloquantes persistantes : quand un message correspondant au type posté arrive dans le système de communications d'un démon, une fonction (callback) est appelée pour effectuer l'action correspondante. Ainsi, le système de communications n'a pas à être spécifiquement en train d'attendre ce message : s'il arrive, le callback correspondant sera appelé.

Group Communications (GRPCOMM) Fonctionnalité essentielle de l'environnement d'exécution, les communications collectives sont implémentées dans le framework GRPCOMM. GRPCOMM fait appel à ROUTED pour choisir par où router les messages, et ainsi envoyer moins de messages. Ce framework utilise les communications de RML.

Inputs/Outputs Forwarding Service (IOF) Les entrées et sorties des processus distants sont transférées vers le processus mpiexec, qui sert de frontale entre l'application et l'utilisateur, par le framework IOF. Les composants disponibles correspondent au rôle particulier des processus, des démons et du mpiexec. Chaque processus (processus MPI ou démon de l'environnement d'exécution) charge celui qui lui correspond à l'initialisation.

Outils

Environment-Specific Services (ESS) Lors de l'initialisation des démons de l'environnement d'exécution et des processus MPI, l'ESS est chargé d'initialiser la partie ORTE selon le rôle du processus. Les différents processus impliqués dans d'exécution d'une application MPI (processus MPI mais aussi démons de l'environnement d'exécution, mpiexec et outils) utilisent les même frameworks pour des raisons de réutilisation de code. Cependant ils ne remplissent pas tous le même rôle et ont donc des utilisations spécifiques de ces frameworks. L'ESS les initialise donc de manière adaptée afin de spécialiser leur usage.

Snapshot Coordination Interface (SNAPC) Ce framework est un outil utilisé uniquement par le système de tolérance aux pannes par points de reprise coordonnés. En-dehors de ce cas il n'est jamais utilisé ni même initialisé. Il est chargé de démarrer une vague de points de reprise et de copier et stocker les points de reprise sur la machine sur laquelle est exécuté le processus mpiexec.

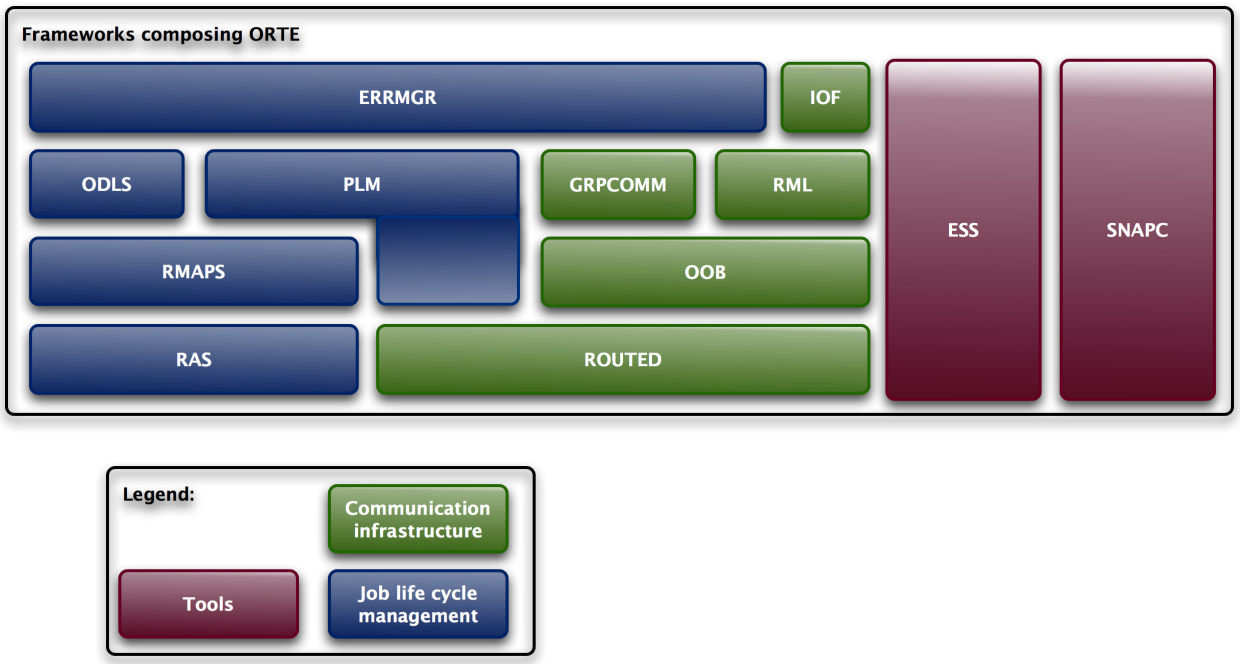


FIGURE 2.2 – *Architecture d'ORTE, organisation des frameworks.*

2.1.2.3 Architecture modulaire

Les composants sont compilés sous forme de bibliothèques dynamiques ou statiques. Les bibliothèques dynamiques sont chargées à l'initialisation de l'application : lors de l'initialisation d'ORTE pour les frameworks d'ORTE, lors de l'initialisation de la bibliothèque MPI (`MPI_Init()`) pour les frameworks d'OMPI. Comme vu ci-avant, certains frameworks d'ORTE sont utilisés par les processus MPI : ils sont alors appelés par la bibliothèque ORTE (`liborte`) et initialisés avant l'initialisation d'OMPI.

L'initialisation d'un framework pour lequel un seul module peut être utilisé à la fois se fait selon l'algorithme décrit par le listing B.1 donné en annexe B. Cet algorithme parcourt la liste des modules disponibles et choisit le celui de plus haute priorité qui a pu être chargé. Ainsi, si un composant ne peut pas être utilisé sur le système, par exemple parce qu'il dépend d'une bibliothèque qui n'est pas disponible, aucun module ne sera initialisé pour ce composant et l'algorithme passera au composant suivant sans le choisir. Les frameworks pouvant utiliser plusieurs modules simultanément chargent tous les modules disponibles, ne sélectionne pas celui de plus haute priorité et ne décharge aucun module qui a pu être initialisé.

De cette manière, les modules non sélectionnés restent en mémoire et utilisent de l'espace inutilement. Or, on cherche plutôt à minimiser l'espace mémoire utilisé par la bibliothèque de communications et l'environnement d'exécution afin d'en laisser le plus possible à la disposition de l'application.

L'utilisateur peut spécifier sur la ligne de commande qui lance son application ou dans les fichiers de configuration d'OpenMPI quels modules il souhaite utiliser ou exclure. Cependant le mécanisme d'origine de sélection des modules effectue celle-ci *après* le chargement et l'initialisation de tous les modules disponibles, en parcourant la liste des modules ouverts et en sélectionnant le ou les module(s) sélectionné(s) ou en excluant ceux que l'utilisateur a demandé de ne pas utiliser.

Amélioration du chargement des composants La première amélioration que j'ai faite dans MCA consiste à décharger les modules non utilisés, comme illustré dans le listing B.2 donné en annexe B. La première partie de la sélection consistant à charger et sélectionner les modules se déroule de la même façon que l'algorithme précédemment en place. Elle est suivie d'un déchargement des modules non utilisés afin de libérer de l'espace mémoire. La deuxième amélioration est une amélioration du choix des composants que l'on tente d'ouvrir, en constituant la liste `liste_des_composants`. L'algorithme précédent ajoutait tous les composants disponibles, les ouvrait tous, et faisait ensuite le tri parmi les composants ouverts. J'ai modifié l'algorithme de construction de cette liste des composants à tenter d'ouvrir pour ne garder que ceux demandés par l'utilisateur, et exclure ceux que l'utilisateur a demandé de ne pas utiliser.

Ces deux optimisations de MCA ont permis de réduire l'utilisation en mémoire d'OpenMPI (y compris lors de l'initialisation d'OpenMPI) et le temps d'initialisation. Pour mesurer cette utilisation mémoire, on peut exécuter une application triviale qui initialise et finalise l'application MPI, en effectuant une pause entre les deux permettant de relever la mesure. Ainsi, l'utilisation de la mémoire du processus correspond uniquement à l'espace utilisé par la bibliothèque MPI : le processus lui-même ne prend pas de place. La taille de l'espace mémoire utilisé par le processus est obtenue dans le système disponible sous Linux dans `/proc`.

L'espace mémoire utilisé a été divisé par environ vingt, passant de 140 Mo à 7 Mo en utilisant la ligne de commande donnée par le listing 2.1 sur une grappe Linux disposant d'un réseau TCP. Cette ligne de commandes a été utilisée avec OpenMPI 1.3a1 obtenu sur le dépôt subversion de développement en juillet 2007. Elle n'est pas valable avec toutes les version d'OpenMPI, certains frameworks étant renommés, supprimés ou créés au fil du développement.

Listing 2.1 – Exemple de ligne de commande pour lancer une application MPI en réduisant l'espace mémoire utilisé par OpenMPI

```
mpiexec --mca btl tcp,self --mca mpool rdma --mca osc rdma \
        --mca pml ob1 --mca rcache rc --mca odls default --mca pls rsh \
        --mca rml oob --mca paffinity linux --mca backtrace none \
        --mca crs ^blcr,self --mca maffinity libnuma --mca memory ^darwin \
        --mca timer linux -n 4 appli
```

Chargement à la demande Une autre modification proposée pour MCA concerne le moment du chargement des modules. OpenMPI charge les modules lors de son initialisation : démarrage des démons pour l'environnement d'exécution, initialisation de la bibliothèque ORTE et de la bibliothèque OMPI pour les processus MPI. Cette initialisation est faite sans condition : ainsi, des processus n'ayant pas besoin d'un framework l'initialiseront quand même, bien qu'ils n'aient pas besoin de l'utiliser. C'est par exemple le cas du PLM, qui n'a besoin d'être chargé que par le processus mpiexec et, dans le cas d'un déploiement en arbre et seulement dans ce cas, les démons de l'environnement d'exécution. En-dehors du déploiement en arbre les démons n'ont pas besoin de l'initialiser, et les processus MPI n'en ont pas besoin non plus.

Le chargement à la demande propose de n'initialiser un framework et de ne charger ses composants qu'au moment où ce framework est utilisé pour la première fois.

Concrètement, on le met en place en initialisant toutes les fonctions de l'interface implémentée par un framework par des talons (*stubs*) qui pointent tous vers un appel à l'initialisation de ce framework et au chargement de ses modules. Ainsi, la première fois qu'une fonction sera appelée, le framework sera initialisé et les fonctions talons seront remplacées par les fonctions du module choisi.

Cette modification de MCA bouleversait trop son fonctionnement et n'apportant pas d'amélioration significative aux performances, elle n'a pas été acceptée dans la distribution d'OpenMPI. Cependant, elle reste une option possible qui pourra être envisagée à nouveau dans le futur dans un logiciel basé sur MCA.

2.2 Plate-forme expérimentale

La plate-forme expérimentale utilisée pour la plupart des mesures de performances est Grid'5000 [41]. Il s'agit d'une grille de calcul destinée à la recherche en informatique, constituée de grappes interconnectées par des fibres noires sur le réseau RENATER.

Un point fort de Grid'5000, qui a été utilisé systématiquement pour les résultats obtenus sur cette plate-forme présentés dans ce document, est la possibilité qu'a chaque utilisateur de déployer une image du système d'exploitation qu'il souhaite utiliser sur les machines qui lui ont été attribuées par le système de réservation. L'utilisateur dispose alors des privilèges du super-utilisateur et peut installer tous les composants logiciels nécessaires à ses expériences sur l'environnement qu'il déploie.

Le système de réservation est OAR [40], couplé avec le système de déploiement Kadeploy [93] et utilisant l'outil de déploiement et d'exécution de commandes à distance TakTuk [54, 123]. Chaque site dispose de son propre système de réservations, et l'outil `oargridsub` permet de coordonner des réservations émises sur plusieurs grappes.

Les grappes utilisées dans cette thèse sont les suivantes :

Grid eXplorer (GdX) , située à Orsay. Cette grappe contient 342 machines équipées chacune de deux micro-processeurs AMD Opteron, certains cadencés à 2,0 GHz et d'autres à 2,4 GHz et de 2 Go de mémoire. Les machines sont toutes équipées de deux cartes réseaux GigaEthernet, et 256 d'entre elles sont reliées à un commutateur Myrinet 10G. Les propriétés des machines comme la fréquence du micro-processeurs ou l'accès au réseau Myrinet peuvent être spécifiées au moment de l'émission de la réservation auprès de OAR.

Paravent , située à Rennes et aujourd'hui hors service (à la retraite). Cette grappe contient 99 machines équipées chacune de deux micro-processeurs AMD Opteron cadencés à 2,0 GHz et de 2 Go de mémoire. Les machines sont toutes équipées d'une carte réseau GigaEthernet, et 66 d'entre elles sont équipées d'une carte InfiniBand 10G [1].

Bordereau , située à Bordeaux. Cette grappe contient 93 machines équipées chacune de deux micro-processeurs AMD Opterons bi-cœurs cadencés à 2,6 GHz et de 4 Go de mémoire. Les machines sont toutes équipées de deux cartes réseaux GigaEthernet.

Pastel , située à Toulouse. Cette grappe contient 80 machines équipées chacune de deux micro-processeurs AMD Opterons bi-cœurs cadencés à 2,6 GHz et de 4 Go de mémoire. Les machines sont toutes équipées d'un carte réseau GigaEthernet.

Helios , située à Sophia Antipolis. Cette grappe contient 56 machines équipées chacune de deux micro-processeurs AMD Opterons bi-cœurs cadencés à 2,4 GHz et de 4 Go de mémoire. Les machines sont toutes équipées d'un carte réseau GigaEthernet et d'une carte Myrinet 2000.

1. Les machines sont en réalité équipées de deux cartes réseau GigaEthernet, mais une seule est utilisée. C'est également le cas des grappes Pastel et Helios, où les machines sont en réalité équipées de quatre cartes.

Chapitre 3

Scalabilité des environnements d'exécution

3.1 Grande échelle et environnement d'exécution

À grande échelle, le lancement et le démarrage d'une application deviennent aussi critique que l'optimisation de l'application elle-même. Si les optimisations des applications et de bibliothèques de communications permettent de réduire le temps d'exécution de l'application elle-même, son temps de déploiement, d'initialisation et de lancement comptent aussi dans le temps total d'exécution.

L'environnement d'exécution est en charge du lancement et de l'initialisation de l'application, et pour cela, il doit se déployer lui-même et mettre en place son infrastructure de communications. Ce déploiement et les fonctionnalités internes de l'environnement d'exécution doivent donc être capables de passer à l'échelle, et ce sur plusieurs plans pour pouvoir être efficace dans les différentes fonctions d'un environnement d'exécution :

- Déploiement de l'environnement d'exécution
- Mise en place de l'environnement d'exécution (construction d'une topologie couvrante)
- Performances des communications collectives sur cette infrastructure, en distribution et en concentration
- Performances des communications point-à-point, notamment pour les sorties (communications des nœuds de calcul vers le processus *mpiexec*)

Le déploiement et l'initialisation d'une application sont comptés dans le temps d'exécution de l'application, mais ce n'est pas du temps *utile*, dans le sens où le calcul n'avance pas pendant ce temps. Ces temps sont donc à prendre en compte dans l'efficacité d'un programme dans sa globalité (et non pas en commençant la mesure après le démarrage de l'application, et en la finissant avant la terminaison de l'application, ce qui ne prend pas en compte le temps machine total utilisé).

Si l'environnement d'exécution ne passe pas à l'échelle et si le temps de déploiement de l'application prend un temps significatif, la rentabilité de l'application est diminuée par le temps perdu par l'environnement d'exécution. Celui-ci doit alors présenter des fonctionnalités efficaces à grande échelle pour ne pas être un frein à la performance et l'efficacité de l'application.

L'introduction de ce chapitre est constituée comme suit. La section 3.1.1 expose un ensemble d'environnements d'exécution et les approches suivies par chacun pour passer à l'échelle. La section 3.1.2 présente le déroulement du lancement d'une application par un environnement d'exécution typique puis les détails de la procédure de lancement mise en place dans OpenMPI, et ses limitations vis-à-vis de la grande échelle. Enfin, la section 3.1.3 expose la méthodologie de mesure utilisée dans cette étude.

3.1.1 Environnements d'exécution pour la grande échelle

On peut distinguer deux orientations dans les environnements d'exécutions pour applications parallèles et les réseaux de couches existants : la première se concentre sur la topologie, la seconde sur la propagation et le lancement.

La bibliothèque de communications pour des réseaux de couches MRNet [10, 147] est directement orientée pour la grande échelle. Elle fournit des fonctionnalités de réduction, diffusion et communications point-à-point permettant de lancer et de supporter des applications distribuées comme des outils de surveillance pour applications parallèles. La topologie de l'infrastructure de communications peut être définie par l'utilisateur dans un fichier de configuration spécifiant les liens entre les processus MRNet. Il ne s'agit cependant pas d'un environnement d'exécution complet, mais d'une bibliothèque de communications : les processus de l'infrastructure MRNet doivent être écrits en utilisant la bibliothèque de communications mais ne sont pas des démons fournis tel quels. De plus, le déploiement de ces processus n'est pas pris en charge et doit être effectué de façon extérieure.

L'environnement de MPICH2, appelé MPD [39], utilise une topologie en anneaux. Cette topologie a été choisie pour des raisons de simplicité, MPD étant à l'origine destiné à tourner en permanence sur les nœuds et devant remplir des impératifs de sécurité. À l'époque où il a été conçu, les grappes courantes étaient composées au plus de quelques dizaines de nœuds, et les plus gros supercalculateurs disposaient d'un système de lancement dédié : cette topologie était alors suffisante car MPD n'avait pas besoin de passer à l'échelle au-delà de quelques centaines de nœuds. Il est aujourd'hui inadapté à de plus grandes échelles.

HARNESS [75, 73] est un environnement d'exécution d'ancienne génération, utilisé par PVM [154] et FT-MPI [74], qui se base sur une topologie d'arbres rendus tolérants au défaillances par un anneau interconnectant les nœuds à chaque niveau de l'arbre [7], qui permet d'utiliser directement des algorithmes efficaces pour les communications collectives. Le but de HARNESS était de supporter des applications parallèles et distribuées en permettant de mettre en place des stratégies de tolérance aux pannes au niveau de l'application.

Selon une autre approche, TakTuk [123, 54] est avant tout un outil de déploiement qui met en place une infrastructure de communications suivant son infrastructure de déploiement en conservant les liens du déploiement. Il s'agit d'un outil multi-fonctions permettant entre autres de déployer des fichiers ou d'exécuter des commandes à distance sur un grand nombre de machines, l'objectif de TakTuk étant la scalabilité, éventuellement sur des architectures hétérogènes. Il est possible de faire circuler des commandes sur cette infrastructure de communications, et une bibliothèque de communications, Taktukcomm, est disponible pour faire communiquer des nœuds TakTuk entre eux. C'est ainsi qu'il est utilisé comme environnement de déploiement par KAAPI [90]. Cependant, KAAPI dispose de son propre environnement d'exécution : TakTuk n'est utilisé que comme lanceur du moteur KAAPI, qui est lui-même en charge du déroulement de l'exécution et de la coordination des fils d'exécution.

Dans cette catégorie, on peut également citer quelques lanceurs d'applications, souvent intégrés à un système de réservation, comme SLURM [108], LSF [1], Torque [2] ou PBS [3]. TakTuk [54] peut de la même façon être déployé de façon persistante, et lancer des commandes sans finaliser l'infrastructure TakTuk en fin d'exécution de la commande. Il s'agit généralement d'un ensemble de démons exécutés de façon persistante sur les machines d'une grappe. La frontale de la grappe exécute un démon particulier servant de contrôleur et qui permet d'émettre et de coordonner des réservations d'une part (rôle du gestionnaire de ressources), et de lancer une commande sur les nœuds attribués à une tâche (rôle du lanceur). Les démons exécutés sur les nœuds de la grappe sont interconnectés selon une infrastructure de communications persistante sur laquelle ils font circuler des commandes pour lancer des applications ou s'assurer de la terminaison d'une application une fois la fin de son temps imparti écoulé. Les démons sont généralement lancés au démarrage des machines, et se connectent au contrôleur sur un port fixe connu. Ce sont principalement des lanceurs, cependant leur infrastructure de communications peut en principe leur permettre de servir d'environnement d'exécution à part entière ; c'est en partie le cas de SLURM avec MPICH [99], MPICH2 [37] et certains de leurs dérivés, où SLURM prend la place de MPD.

Sur les machines à très grande échelle, comme Blue Gene/L [89], un environnement d'exécution spécifique est parfois implémenté spécialement pour la machine. Dans le cas de Blue Gene/L [3], les nœuds de calcul ne peuvent pas exécuter plusieurs processus en même temps, donc ils ne peuvent pas exécuter de démon pour l'environnement d'exécution. Un démon, basé sur SLURM, est exécuté sur les nœuds d'entrées-sorties et sert de relais entre les nœuds de calcul et la frontale. C'est lui qui envoie sur le réseau de contrôle des requêtes pour exécuter directement des appels systèmes sur les nœuds de calcul, comme le lancement, la coordination et la terminaison des processus exécutés sur les nœuds de calcul. Cette structure hiérarchique permet d'obtenir un

1. LSF : Load Sharing Facility. `http://www.platform.com/Products/platform-lsf`
2. Torque Resource Manager. `http://www.clusterresources.com/products/torque-resource-manager.php"`
3. PBS : Portable Batch System. `http://www.pbsgridworks.com/`

lancement scalable des applications dans l'environnement spécifique de Blue Gene/L.

Dans ce chapitre, les résultats présentés ont été mis en place dans OpenMPI [85]. Le choix a été fait d'utiliser OpenMPI principalement du fait de son architecture : chaque fonctionnalité à remplir par l'environnement d'exécution est isolée dans un framework, et chaque framework est implémenté d'une façon différente dans les composants. Par exemple, modifier la façon dont l'environnement d'exécution se déploie sur les nœuds de calcul revient à ré-implémenter le framework chargé de cette tâche.

3.1.2 Déroulement du lancement d'une application

Le lancement d'une application nécessite le déploiement préalable de l'environnement d'exécution et l'établissement de l'infrastructure de communications de celui-ci. Cette infrastructure de communications est destinée à permettre la circulation de messages hors-bande nécessaires au support de l'application, notamment son lancement et la mise en contact de ses processus.

Cette première étape nécessite le lancement des démons de l'environnement d'exécution sur les nœuds qui seront utilisés pour le calcul, et l'établissement de connexions entre eux de manière à établir une topologie connexe d'arbre couvrant sur l'ensemble des démons de l'environnement d'exécution.

Une fois établie, cette infrastructure de communications peut alors être utilisée pour propager la commande de lancement de l'application. Les processus de l'application sont lancés par les démons de l'environnement d'exécution, généralement en se clonant.

La dernière étape de lancement d'une application consiste à mettre en relation les processus de l'application les uns avec les autres. Pour des raisons de passage à l'échelle, les connexions elles-mêmes sont souvent établies à la demande au moment d'effectuer la première communication entre deux processus. Cette mise en relation lors de l'initialisation consiste alors souvent à diffuser au sein de l'application les informations de connexion des processus. Bien qu'il soit possible de découvrir les informations de connexion d'un processus distant au moment d'établir une connexion avec lui, comme nous allons le voir dans le chapitre 5, les implémentations MPI généralistes procèdent le plus souvent en les propageant lors de l'initialisation de l'application. L'environnement d'exécution ne faisant pas d'hypothèse sur les schémas de communications de l'application, tout processus peut potentiellement communiquer avec chacun des autres processus. Chacun d'entre eux dispose donc des informations de connexion de l'intégralité des autres processus. L'ensemble de ces informations est mis à leur disposition au moment de l'initialisation plutôt qu'à la demande, au moment où la connexion doit être établie : ainsi, le coût de cette mise à disposition (un ensemble de communications hors-bande) est payé une fois pour toutes au moment du démarrage.

Une autre solution consiste à utiliser un même port réseau, défini par la configuration de la bibliothèque de communications. En connaissant les noms des machines utilisées par l'application et la correspondance entre les processus et ces machines, il n'est alors pas nécessaire d'effectuer d'échange d'informations de connexion entre les processus. Cependant, cette solution n'est pas suffisamment générique pour être envisageable dans le cas général.

Le lancement et l'initialisation d'une application est schématisé par la figure 3.1. Dans OpenMPI, le déploiement de l'environnement d'exécution est initié par le processus `mpiexec`. Ce processus fait partie de l'environnement d'exécution, et est vu par lui comme un démon particulier appelé *Head Node Process* (HNP) pour son rôle de frontale entre l'utilisateur et l'application. La phase d'initialisation de l'environnement d'exécution est appelée `orte_init`, à l'image de l'initialisation de la bibliothèque MPI, `MPI_Init`.

La version d'ORTE incluse dans OpenMPI 1.2 procède par phases. Chacune des phases se termine par une synchronisation de tous les processus auprès du HNP. Il est alors possible d'utiliser ces phases pour découper le temps de lancement d'une application et de déterminer où le temps est principalement passé.

La figure 3.2 montre le découpage du temps de lancement d'une application MPI effectuant uniquement une initialisation de la bibliothèque MPI et une finalisation (`MPI_Init` puis `MPI_Finalize`), banc de test le plus complet pour mesurer l'efficacité d'un environnement d'exécution à déployer et initialiser une application MPI. Cette mesure a été effectuée sur la grappe GdX de Grid'5000, en utilisant un processus par machine.

On constate que deux phases dominent : le lancement des démons, et la première phase, dite "*stage 1*". Cette phase correspond à la connexion des démons vers le HNP pour établir la topologie connexe de l'environnement d'exécution, au lancement de l'application et à l'échange des informations de connexion des processus MPI. On voit également que le `MPI_Finalize` a un temps négligeable devant le temps total d'exécution, de même que les

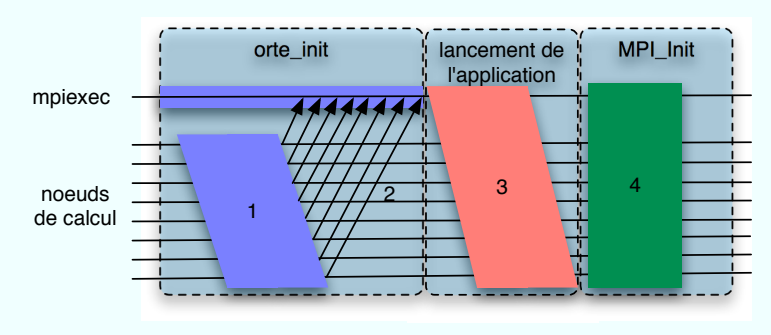


FIGURE *3.1 – Déroulement du lancement synchrone d'une application avec OpenMPI : déploiement de l'environnement d'exécution (1), récupération des informations de connexion et mise en place de l'infrastructure de communications de l'environnement d'exécution (2), lancement de l'application (3), échange des informations de contact entre les processus MPI (4).*

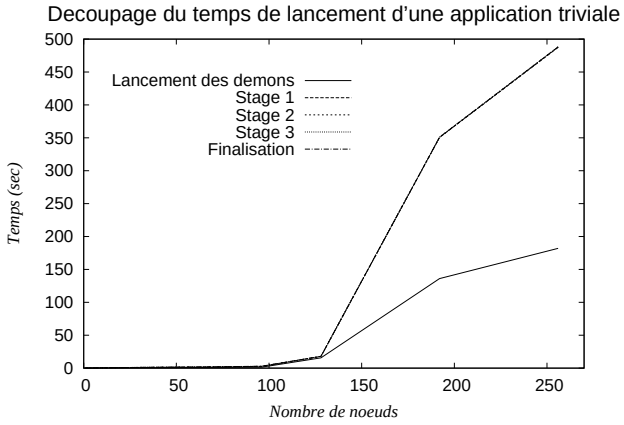


FIGURE *3.2 – Découpage du temps de lancement d'une application MPI en phases. Les phases de lancement des démons et de "stage 1" sont largement dominantes par rapport aux autres, qui apparaissent confondues avec la courbe de stage 1.*

phases 2 et 3, qui contiennent des initialisations locales mais aucune communication.

Les points sur lesquels l'effort de passage à l'échelle doit se concentrer sont donc le déploiement de l'environnement d'exécution lui-même, la mise en place de son infrastructure de communications et l'échange des informations de communications de l'application MPI. Ces trois opérations sont effectuées de façon séparée (et même parfois en utilisant une application tiers pour le déploiement). Cependant, il est possible d'intégrer plus fortement le déploiement de l'environnement d'exécution et la construction de l'infrastructure de communications et d'éviter certaines synchronisations entre les processus.

Ce chapitre présente des améliorations que j'ai proposées pour améliorer le passage à l'échelle de ORTE, l'environnement d'OpenMPI. Certaines d'entre elles figurent aujourd'hui dans la distribution d'OpenMPI : l'arbre de lancement présenté en section 3.2.1 que j'avais initialement implémenté dans la première version d'ORTE a été porté dans la nouvelle version par Ralph Castain. J'ai implémenté un premier prototype d'une construction efficace pour l'infrastructure de communications dans la première version d'ORTE, qui a ensuite été adapté pour bénéficier du framework de routage dans le déploiement et la construction de l'infrastructure de communications comme présenté dans la section 3.2.2 et porté dans la version officielle d'OpenMPI par George Bosilca. Enfin, il a été tenu compte de mes travaux sur les communications collectives et la hiérarchisation engendrée par la gestion de plusieurs processus par machine dans la refonte de celles-ci et la suppression de certaines topologies occasionnées par le passage de ORTE1 à ORTE2.

```
#include <mpi.h>
#include <stdlib.h>

int main( int argc, char **argv ) {
  MPI_Init( &argc, &argv );
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

```
BEGIN=‘date +%s%N‘
mpiexec --machinefile machinefile -n $NP appli_test
END=‘date +%s%N‘
TOTAL=$(($END-$BEGIN))
echo $NP"  "$TOTAL >> time.dat
```

FIGURE *3.3 – Prise de mesure d'un lancement d'application MPI.*

3.1.3 Méthodologie pour les mesures de performances

On peut définir trois bancs de test successifs, pour mettre en évidence chacun une fonctionnalité particulière de l'environnement d'exécution : le déploiement de l'environnement d'exécution et le lancement d'une application, l'agrégation de communications vers un point (le processus mpiexec), et l'initialisation d'une application MPI.

Le premier consiste à exécuter une application qui ne fait rien : aucun calcul, et aucune communication ni aucun affichage. Par exemple, l'application `/bin/true`, présente sur la plupart des systèmes Unix et ses clones, est un bon candidat. Le temps d'exécution du `mpiexec` correspond alors au temps de déploiement de l'environnement d'exécution, sa mise en place et au lancement de l'application.

Une fois le temps de lancement isolé, des bancs de test plus avancés peuvent mettre en évidence d'autres fonctionnalités de l'environnement d'exécution. Son infrastructure de communications peut être sollicitée en effectuant des affichages. En effet, la transmission des signaux et des entrées-sorties de l'application et vers elle fait partie du rôle de l'environnement d'exécution. Le cas extrême consiste à faire communiquer tous les processus de l'application vers le `mpiexec` en leur faisant faire un affichage. On peut leur faire effectuer un affichage de taille fixe, par exemple en utilisant `hostname` ou `uptime`, ou faire varier la longueur de la chaîne de caractères affichée.

Le dernier banc de test met en évidence le temps d'initialisation d'une application MPI. Cette initialisation consiste à rendre possible les communications entre les processus MPI, c'est-à-dire, échanger leurs informations de contact. Cette opération est effectuée par la routine MPI `MPI_Init`. On peut donc utiliser un programme MPI qui effectue un `MPI_Init` immédiatement suivi d'un `MPI_Finalize`, et se termine, comme illustré figure 3.3. Le temps d'exécution du `MPI_Finalize` est négligeable devant le `MPI_Init`, comme vérifié dans ce qui suit. Ce banc de test sera dans la suite désigné comme “init/finalize”.

3.2 Lancement d'une application

Le déploiement de l'environnement d'exécution est fait en contactant un démon s'exécutant sur les machines distantes pour lancer le démon de l'environnement d'exécution. Dans le cas des lanceurs spécialisés comme SLURM [108] ou LSF [4], ces démons sont exécutés en permanence sur les nœuds de la grappe. Ces nœuds sont généralement accessibles par les utilisateurs normaux uniquement en utilisant les communications entre ces démons, qui doivent obligatoirement passer par une machine frontale où ils soumettent leurs tâches dans l'ordonnanceur associé. TakTuk [54] utilise SSH pour s'auto-déployer, et peut également être déployé en mode persistant et faire circuler des commandes entre les démons TakTuk.

Dans le cas d'un lanceur utilisant des démons déployés au préalable pour lancer un processus de l'environnement d'exécution sur chaque machine impliquée dans le calcul, l'ensemble des démons du lanceur peut alors être vu comme un environnement d'exécution de l'environnement d'exécution, puisqu'il est exécuté sur les nœuds sur lesquels l'environnement d'exécution de l'application MPI peut être exécuté, qu'il permet de lancer les démons qui le constituent, et qu'il dispose d'une infrastructure de communications permettant de le supporter.

Dans le cas présent, nous nous intéressons au lancement utilisant SSH ou RSH. Il s'agit de la situation la plus générale, ne nécessitant pas de configuration particulière des nœuds de calcul. De plus, ce mode de lancement permet de contrôler chaque commande de lancement pendant le déploiement même de l'environnement

4. LSF : Load Sharing Facility. `http://www.platform.com/Products/platform-lsf`

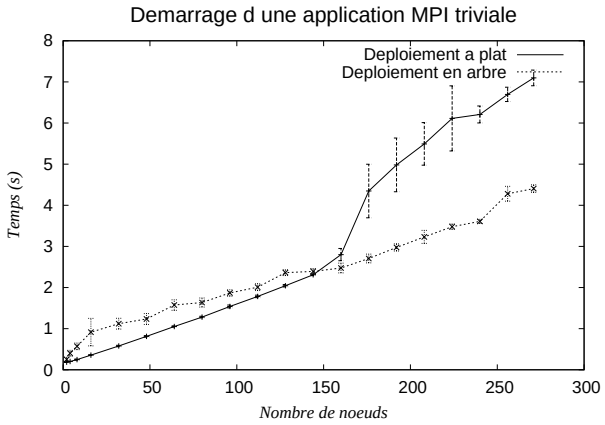


FIGURE *3.4 – Lancement d'une application MPI triviale (init/finalize) selon un arbre binomial et lancement séquentiel dans ORTE1. Toutes les informations sont transmises aux démons de l'environnement d'exécution lors de leur lancement.*

d'exécution, tandis que les lanceurs externes utilisent une commande construite une seule fois et exécutée pour tout le déploiement [5].

Dans un premier temps, nous allons nous concentrer sur la topologie de déploiement seule, indépendamment de la construction de l'infrastructure de communications, qui sera vue plus tard (section 3.2.2). Dans cette section, l'infrastructure de communications est établie en connectant chaque démon directement au HNP après son lancement.

3.2.1 Topologie de déploiement

Le déploiement de l'environnement d'exécution commence par le lancement d'un démon par nœud de calcul. En utilisant SSH ou RSH, il s'agit de contacter tous les nœuds et de leur envoyer une commande à exécuter : en quelque sorte, cette opération revient à une diffusion. Dans un modèle un-port et pour un nombre de nœuds qui est une puissance de deux, l'algorithme optimal pour effectuer une diffusion est un arbre binomial. Il est quasi-optimal pour un nombre qui n'est pas une puissance de deux.

À l'inverse, un déploiement centralisé au niveau du HNP ne peut pas passer à l'échelle car il est séquentiel. De plus, l'établissement de l'infrastructure de communications, en effectuant des connexions vers le HNP, subit la concurrence de ce lancement : le moteur de communications de ORTE ne commence à accepter les communications qu'une fois la campagne de SSH qui lance les démons distants est terminée.

Dans la version d'ORTE incluse dans OpenMPI 1.2, tous les démons reçoivent la liste des machines sur lesquelles lancer les démons au moment où ils rejoignent l'unfrastructure de communications. Chacun peut alors regarder s'il doit lancer d'autres démons, pour mettre en place un arbre. Les performances de ce déploiement en arbre avec celles du déploiement à plat sont présentées figure 3.4.

On constate une amélioration significative de la scalabilité avec le lancement en arbre, dont le temps de lancement augmente moins vite que le lancement séquentiel. Le temps du lancement séquentiel augmente plus vite que linéairement, alors que celui du lancement en arbre augmente moins vite. De plus, on constate une plus grande stabilité du lancement en arbre, qui présente un écart type réduit.

Cependant, le fait de transmettre la liste des machines aux démons en les lançant à distance, en la passant sur la ligne de commandes, peut rapidement devenir impossible. Une refonte du fonctionnement de l'environnement d'exécution a été intégrée à partir de la version 1.3 d'OpenMPI. Dans la suite, on appelle ORTE2 l'environnement d'exécution lorsqu'il s'agit de fonctionnalités propres à cette version, et ORTE1 lorsqu'il s'agit de fonctionnalités propres à la version incluse dans OpenMPI 1.2.

À partir d'OpenMPI 1.3, ORTE2 procède différemment d'ORTE1 : la liste des machines n'est transmise aux démons de l'environnement d'exécution qu'après leur lancement, si besoin. En particulier, dans le cas du

5. Bien que TakTuk, par exemple, permette de lancer une commande qui n'est pas la même pour chaque nœud, celle-ci est construite en début de déploiement et n'est pas modifiée pendant son déroulement.

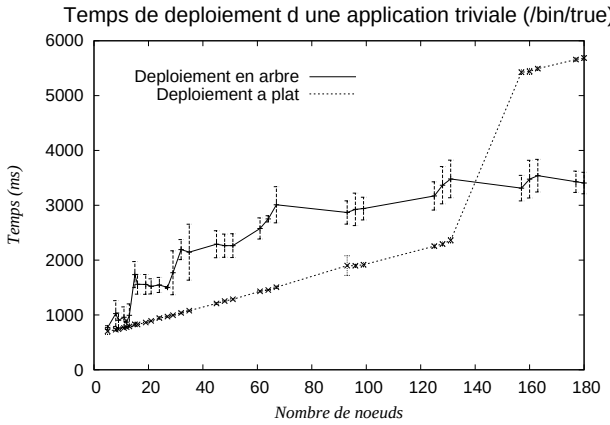


FIGURE *3.5 – Lancement d'une application triviale (/bin/true) selon un arbre binomial et lancement séquentiel dans ORTE2. Les informations sont transmises aux démons de l'environnement d'exécution après leur lancement, par le HNP : tous les démons se connectent directement au HNP.*

lancement séquentiel, les démons (hors HNP) n'ont pas besoin de cette liste : ils ne la reçoivent donc pas. Dans le cas du lancement en arbre, le démons la reçoivent une fois l'infrastructure de communications partiellement mise en place, par une communication avec le HNP.

Cette approche centralise les communications sur le HNP : il ne lance plus tous les démons distants, mais tous se connectent à lui une fois lancés, et il doit leur envoyer à tous, un par un, la liste des machines pour qu'ils puissent lancer leurs fils dans l'arbre de déploiement. Les performances de ce déploiement sont présentées figure 3.5, et comparées avec le lancement séquentiel décrit ci-avant.

On constate que les performances du lancement à plat sont améliorées par rapport au lancement séquentiel précèdent (figure 3.4) jusqu'à environ 130 processus, du fait de cette absence de transmission d'information. Cependant, le déploiement séquentiel montre toujours une forte dégradation de ses performances à partir d'environ 130 processus.

Le déploiement en arbre montre une meilleure scalabilité, mais de moins bonnes performances que le déploiement à plat avant la dégradation des performances de celui-ci. Cela s'explique par la communication supplémentaire du HNP vers les démons pour leur transmettre la liste des machines à utiliser, nécessaire à la continuation de l'arbre. De plus, on constate une plus grande variabilité des performances du lancement en arbre : en effet, tous les démons se connectant au même point (le HNP) et sollicitant une communication, une concurrence se met en place au niveau des communications réseau du HNP, introduisant une variabilité importante d'une exécution à une autre. Cette variabilité est moins importante dans le cas du déploiement à plat, du fait du caractère séquentiel de ce mode de déploiement, que pour le déploiement en arbre, qui introduit du parallélisme au niveau des lancements des démons et donc au niveau des arrivées des connexions sur le HNP.

Un déploiement en arbre élimine le point central de contention constitué par un déploiement séquentiel effectué uniquement par le HNP, la centralisation de la mise en place de l'infrastructure de communications concentre des connexions simultanées vers un même processus, qui doit alors les accepter et y répondre en renvoyant la liste des machines à utiliser pour la suite du déploiement.

3.2.2 Mise en place de l'infrastructure de communications

Si la mise en place de l'infrastructure de communications s'effectue de façon centralisée, tous les démons de l'environnement d'exécution se connectent au même moment vers un même point, qui se trouve en plus être le point de départ des connexions SSH pour lancer les démons qui n'ont pas encore été déployés.

Une fois lancé, chaque démon rejoint l'environnement d'exécution en se connectant à un processus dont il connaît les informations de contact et, dans le cas d'un déploiement en arbre, celui-ci lui obtient ses informations de connexion et envoie en réponse la liste des machines utilisées afin de lui permettre de poursuivre le déploiement. Si le point de contact est un unique processus pour tous les démons de l'environnement d'exécution,

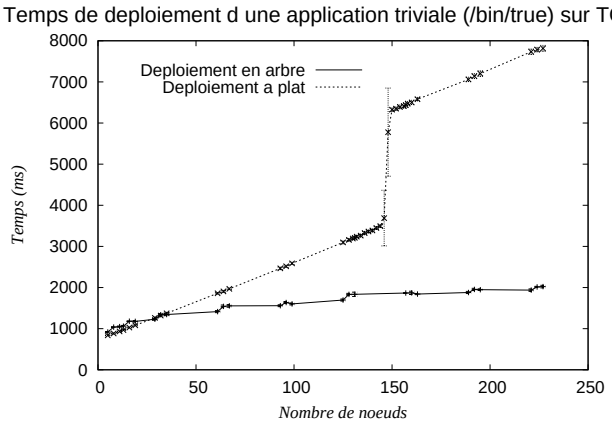


FIGURE *3.6 – Lancement d'une application triviale synchrone et asynchrone avec correspondance de l'arbre de routage des communications hors-bande selon l'arbre de déploiement de l'environnement d'exécution.*

celui-ci doit alors accepter et gérer les connexions de chaque nouveau démon.

Le HNP subit alors une tempête de connexions arrivant des démons qui viennent d'être lancés, et des paquets peuvent être perdus dans des tampons trop pleins, ou retardés. Le lanceur d'OpenMPI, en particulier, commence à traiter les connexions entrantes une fois la campagne de SSH terminée, ou après un certain nombre de démons lancés (configurable par l'utilisateur).

On peut alors utiliser l'arbre de routage comme topologie de déploiement : ainsi, l'arbre établit lors du déploiement sera utilisé pour la transmission des messages dans l'environnement d'exécution. Chaque démon se connecte alors à son parent dans l'arbre de déploiement, et c'est celui-ci qui lui envoie la liste des machines afin qu'il puisse poursuivre le déploiement. Le routage des messages se fera alors le long de cet arbre.

Dans le cas du lancement à plat, la construction de l'arbre de routage se fait après la phase de lancement. Une fois tous les démons connectés au HNP, celui-ci dispose de toutes leurs informations de contact, et peut les transmettre aux démons qui en ont besoin pour établir une connexion.

Les performances de ce lancement en arbre avec établissement de l'arbre de routage sont présentées figure 3.6. Ces mesures ont été réalisées en utilisant l'émulation TCP fournie par le pilote MX pour les réseaux Myrinet, en utilisant le réseau Myricom 10g de la grappe GdX. Ce réseau utilisant un seul commutateur disposant de 256 ports, la topologie physique de réseau est plate, contrairement au réseau Ethernet qui utilise plusieurs commutateurs. Le déploiement n'est alors pas sujet à des hétérogénéités de communications dues au passage par un nombre variables de commutateurs selon les nœuds communiquant.

On constate, par rapport au lancement avec établissement centralisé de l'infrastructure de communications présenté figure 3.5, une meilleure performance du lancement en arbre et une variabilité beaucoup plus faible. Le déploiement en arbre est plus rapide que le déploiement à plat, sauf à petite échelle où le gain de l'arbre ne suffit pas à compenser complètement la communication supplémentaire qu'il nécessite. On constate bien, par ailleurs, les marches correspondant aux puissances de deux sur le déploiement en arbre.

Le déploiement à plat présente un saut de trois secondes aux alentours de 150 nœuds : en prenant une mesure tous les deux nœuds, on constate que ce saut est très net. De plus, les mesures effectuées pendant ce saut présentent une variabilité forte, au contraire des mesures effectuées juste avant ou juste après le saut. En comptant les paquets SYN envoyés par chaque nœud vers le HNP, on constate qu'à partir de cette taille certains nœuds en envoient un de plus. Les nœuds qui envoient un paquet supplémentaire correspondent à un démon de rang élevé, et sont de plus en plus nombreux lorsque le nombre total de nœuds augmente.

Le HNP subissant une tempête de connexion alors qu'il doit effectuer des communications simultanément aux connexions arrivant (lancement des démons suivants, envoi des listes de machines à ceux qui se sont connectés), certaines demandes d'établissement de connexions sont retardées ou perdues. Le tampon contenant les connexions TCP "semi-ouvertes" (dont la triple poignée de mains n'a pas été achevée), côté serveur, a une taille limite définie en paramètre de l'appel `listen()`, mais ne pouvant pas être supérieure à `SOMAXCONN` (fixé à 128 pour les systèmes Linux et BSD). Si ce tampon est plein, les paquets SYN envoyés par les processus essayant

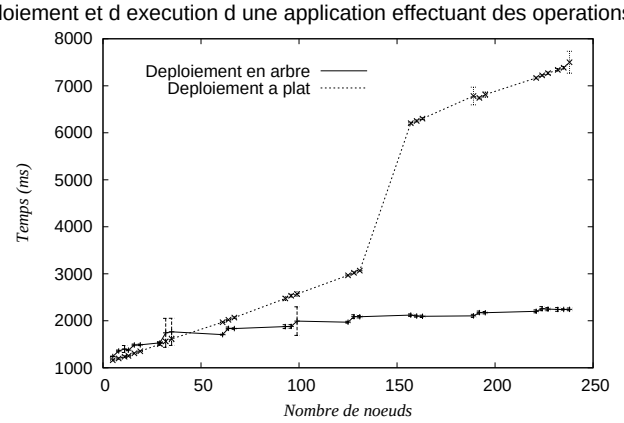


FIGURE *3.7 – Lancement d'une application effectuant des opérations de sorties (affichages) avec correspondance de l'arbre de routage des communications hors-bande selon l'arbre de déploiement de l'environnement d'exécution.*

d'établir des connexions sont perdus.

Les démons doivent alors ré-émettre des paquets SYN, ce qui est effectué, sous Linux, après trois secondes. On constate sur la courbe que ces trois secondes correspondent bien à l'écart de performances avant et après ce saut. Comme le déploiement à plat attend que tous les démons de l'environnement d'exécution se soient connectés au HNP avant de passer à la phase suivante du lancement, tout le déploiement est retardé par ces démons qui doivent ré-émettre l'initialisation de leur connexion avec le HNP.

Une topologie de communications en arbre n'introduisant pas de point central de ce type, et n'engendrant donc pas de telle tempête de connexions, on peut s'attendre à ce qu'à plus grande échelle il ne subisse pas ce problème.

La même mesure peut être effectuée en utilisant le réseau TCP et le banc de test *hostname*, pour vérifier les performances de l'infrastructure de communications en tant que réseau de concentration. La figure 3.7 compare le temps de déploiement et d'exécution d'une application effectuant des opérations de sortie (affichage sur la sortie standard). On constate que, pour les deux déploiements, ces affichages n'affectent pas de manière significative les performances. En effet, pour les deux topologies de déploiement, l'infrastructure de communications établit une topologie en arbre selon laquelle les communications hors-bande sont routées.

Le gain en performances lié à l'établissement de l'arbre de routage se fait lors du déploiement de l'environnement d'exécution. Une fois l'application lancée, dans les deux cas les messages sont routés selon un arbre défini par le routage. C'est donc la topologie suivie par le routage qui est déterminante pour les performances des communications hors-bande.

3.3 Synchronisme du lancement

Les phases successives de lancement à plat d'une application par ORTE sont effectuées de façon synchrone : une phase ne démarre que si la phase précédente est terminée, comme illustré par la figure 3.1. Le lancement en arbre permet d'établir un pipeline entre les phases, et de ne pas avoir à attendre la fin d'une phase pour déclencher la suivante.

3.3.1 Synchrone

Un lancement d'application centralisé peut être réalisé par phases successsives (figure 3.1) :
- Lancement des démons distants, par une campagne de lancements distants ;
- Connexion de chaque démon distant directement au démon central ; les connexions des démons distants ne sont acceptées qu'à la fin de chaque campagne de lancements, la taille d'une campagne de lancements étant réglable ;

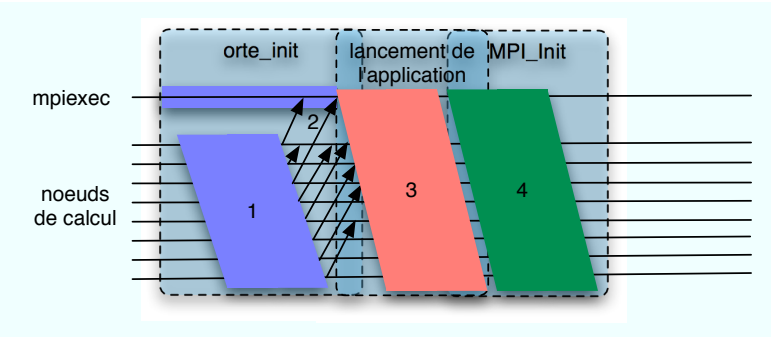


FIGURE *3.8 – Déroulement du lancement asynchrone d'une application : déploiement de l'environnement d'exécution (1), récupération des informations de connexion et mise en place de l'infrastructure de communications de l'environnement d'exécution (2), lancement de l'application (3), échange des informations de contact entre les processus MPI. Chaque phase se fait de façon asynchrone, sans attendre la fin de la phase suivante, contrairement au déploiement synchrone.*

- Une fois *tous* les démons connectés au démon central, lancement de l'application par la diffusion d'une commande ;
- Une fois *tous* les processus de l'application lancés, initialisation de l'application par une opération collective bloquante de type barrière.

En considérant la figure 3.1 comme un diagramme de Gantt, on voit qu'il existe entre chaque phase des temps d'attente pendant lesquels des démons ou des processus ne font rien et attendent que les autres processus aient terminé la phase courante.

3.3.2 Asynchrone

Le lancement en arbre permet d'établir un pipeline entre les démons de l'environnement d'exécution et de superposer le début d'une phase avec la fin de la phase précédente. En rendant le lancement asynchrone, on supprime une partie des temps d'attente entre les phases.

La figure 3.8 illustre comment le lancement d'une application peut être rendu asynchrone :

- Les démons de l'environnement d'exécution sont lancés selon un arbre ;
- Chaque démon se connecte à son parent dans l'arbre de déploiement, mettant ainsi en place l'infrastructure de communications ;
- Dès que le premier étage de l'arbre de déploiement est lancé, la commande de lancement de l'application est propagée entre les démons de l'environnement d'exécution ;
- L'initialisation de l'application est faite par une opération collective non bloquante.

3.3.2.1 Lancement de l'application

Le lancement de l'application est effectué en diffusant une commande dans l'environnement d'exécution. La topologie de l'infrastructure de communications correspondant à l'arbre de routage, la commande est diffusée le long des connexions mises en place lors du déploiement de l'environnement d'exécution.

Il n'est donc pas nécessaire d'attendre que tous les démons soient lancés pour initier la diffusion de la commande de lancement de l'application : seul le premier étage de l'arbre doit être lancé. Une fois que tous les démons devant se connecter à lui sont lancés, le HNP peut passer à la phase suivante et commencer le lancement de l'application.

Le lancement des démons prenant plus de temps que la diffusion d'un commande, il est possible que la diffusion de la commande doive être interrompue au cours de sa descente de l'arbre, qui est alors en cours de construction. Le message est alors mis en attente par le démon devant le transférer à ses fils, qui l'enverra une fois ceux-ci lancés.

On peut ainsi lancer l'application le long de l'arbre directement à la suite du déploiement de l'environnement d'exécution.

3.3.2.2 Initialisation de l'application

L'initialisation de l'application consiste principalement à rendre possible la mise en contact des processus les uns avec les autres. Concrètement, il s'agit de mettre à disposition de chaque processus les informations de communication de tous les autres processus de l'application : cette opération correspond à une communication collective de concaténation avec mise à disposition du tampon résultant auprès de tous les processus participants (équivalent à un *MPI_ Allgather*).

Dans ORTE, cette opération est implémentée en effectuant une concaténation (*MPI_ Gather*) vers le HNP puis une diffusion (*MPI_ Bcast*). L'implémentation de cette opération sera discutée dans la suite (section 3.4).

L'opération de concaténation peut commencer dès qu'un processus est lancé : chaque processus envoie ses informations de communications au HNP dès qu'il est lancé, quel que soit l'état des autres processus. L'arbre de routage correspondant à l'arbre de déploiement, chaque démon envoie son message vers le HNP en passant par des démons qui sont déjà au moins aussi avancés que lui. Contrairement aux algorithmes de concaténation en arbre traditionnels, chaque démon envoie son tampon vers la racine de l'opération sans attendre le tampon de ses fils dans cet arbre, ici encore pour ne pas à voir à attendre ses fils.

Il demeure cependant une source de synchronisme au moment du passage à la deuxième phase de cette opération collective, puisque la racine de la concaténation doit attendre que tous les démons lui aient envoyé leur tampon pour passer à la diffusion du tampon résultant.

Cette opération peut être rendue *non-bloquante* : l'application peut poursuivre son exécution avant qu'elle soit terminée. Si le processus ne dispose pas des informations de connexion au moment d'effectuer une communication, il attend qu'elle soit terminée pour l'effectuer. Ainsi, l'échange des informations de communications entre les processus peut être superposé avec le début de son exécution. Les communications hors-bande sont implémentées dans OpenMPI pour les réseaux TCP uniquement ; cependant, l'opération collective est effectuée par le démon de l'environnement d'exécution, qui met a disposition de ses processus MPI locaux le tampon résultant : les communications peuvent donc bien progresser pendant que l'application effectue des calculs.

3.4 Topologie de l'infrastructure de communications

La topologie de communications est importante pour le fonctionnement de l'environnement d'exécution et les performances des communications hors-bande. L'infrastructure de communications doit être efficace en tant que réseau de distribution (pour transmettre des commandes ou des informations à tous les processus de l'application), en tant que réseau de concentration (pour aggréger des données en provenance des processus) et, plus marginalement, en point-à-point.

Dans ORTE1, des communications point à point peuvent être établies entre deux démons de l'environnement d'exécution. Il n'y a pas de topologie de routage à proprement parler : les communications entre deux démons de l'environnement d'exécution sont effectuées directement en point-à-point. La seule topologie construite de façon fixe par l'environnement d'exécution est celle suivie par les communications collectives.

La possibilité d'établir une connexion point-à-point avec n'importe quel autre démon implique le fait de connaitre l'intégralité de leurs informations de communications. Pour permettre celà, dans ORTE1 le processus mpiexec concatène l'ensemble des informations de communications des démons lorsqu'ils se connectent à lui pour rejoindre l'infrastructure de communications et les diffuse à l'ensemble des démons. Cette diffusion implique un coût supplémentaire lors du lancement de l'environnement d'exécution.

ORTE2 impose le même arbre de routage aux communications collectives et point-à-points. C'est-à-dire que la topologie de routage sert de topologie aux opérations collectives. Cette approche permet de tirer parti de l'infrastructure de communications créée lors du déploiement de l'environnement d'exécution pour toutes les communications hors-bande.

3.4.1 Communications collectives

Seules trois routines de communicatons collectives sont utilisées dans l'environnement d'exécution :

– *une barrière*, qui assure la synchronisation de tous les processus. Il peut s'agir de s'assurer que tous les processus sont bien présents à un point de synchronisation (par exemple à la fin du lancement), d'attendre tous les processus à la fin de l'exécution (certains environnements d'exécution s'assurent que tous les

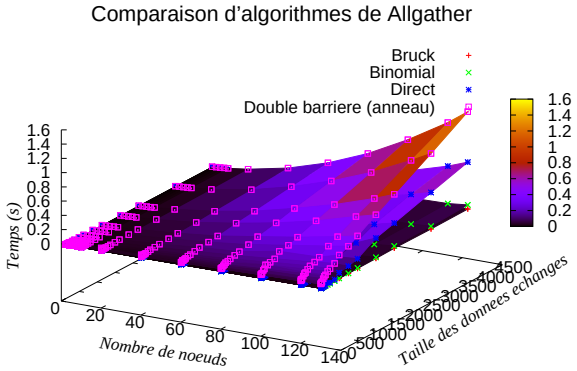


FIGURE *3.9 – Comparaison d'algorithmes de concaténation avec redistribution du résultat en vue de choisir l'algorithme le plus efficace pour l'environnement d'exécution.*

processus ont terminé leur exécution avant de se finaliser), ou pour introduire des points de synchonisation pendant l'exécution (par exemple pour des besoins de tolérance aux pannes) ;

- *une concaténation avec redistribution du résultat* (allgather), qui permet de rassembler des données mises à disposition par chaque processus et de les rendre disponibles auprès de tous les processus. Cette opération est typiquement appelée pour échanger les identifiants de connexion des processus entre eux ;
- *une diffusion*, qui permet de transmettre une commande à tous les processus de l'application ou de l'environnement d'exécution. Par exemple, la commande de lancement de l'application est propagée au sein de l'environnement d'exécution au moyen d'une diffusion. De même, un signal à destination de tous les processus de l'application est propagée dans l'environnement d'exécution par une diffusion, et les démons transmettent à leurs processus locaux.

La barrière équivaut à un allgather où les données transmises seraient nulles. L'algorithme utilisé sera donc le même pour ces deux opérations. Afin d'utiliser l'algorithme le plus efficace pour effectuer cette communication, nous avons comparé les performances de plusieurs algorithmes en utilisant la bibliothèque OCC [6] [135]. OCC permet d'évaluer des algorithmes d'opérations collectives en proposant une implémentation en MPI d'un certain nombre d'algorithmes courants et un framework de prise de mesure. Il s'agit d'un outil d'évaluation d'algorithmes afin de permettre de choisir celui qui sera le mieux adapté à la situation.

Nous avons utilisé l'implémentation de l'algorithme de Bruck [34] et d'une concaténation suivie d'une diffusion selon un arbre binomial. De plus, nous avons implémenté un anneau double et un algorithme direct où le schéma de communication consiste à faire communiquer tous les processus les uns avec les autres. La figure 3.9 présente le résultat de cette comparaison. Les mesures ont été effectuées sur la grappe GdX de Grid'5000, qui a été présenté dans la section 2.2 du chapitre 2.

On constate que l'anneau double, utilisé par MPD [39], est l'algorithme qui présente les moins bonnes performances. En effet, si N désigne le nombre de processus impliqués, il nécessite l'envoi de $2N$ messages pour effectuer deux fois le tour de l'ensemble des processus, et n'offre pas de possibilité d'effectuer des communications simultanément les unes aux autres : un processus n'envoie son message à son voisin de droite qu'après avoir reçu le message de son voisin de gauche, et il reste bloqué dans l'appel jusqu'à réception du second message (deuxième tour de l'anneau). L'algorithme direct implique l'envoi de $2(N-1)$ messages : tous les processus envoient leur donnée vers un processus central (à la manière d'une concaténation), puis une fois que celui-ci à reçu les données de tous les processus il envoie le tampon résultant de la concaténation à tous les processus l'un après l'autre. Dans un modèle un-port, ces envois de messages ont lieu séquentiellement ; cependant, il est généralement possible en pratique d'envoyer plusieurs messages en même temps (remplissage de la bande passante). L'algorithme linéaire envoie $2(N-1)$ messages en $2(N-1)$ étapes dans un modèle un-port. Dans un modèle N-ports, les étapes de transmissions de messages peuvent être superposées : il y a *au plus* $2(N-1)$ étapes. Cette possibilité de superposition rend cet algorithme plus performant que l'anneau double.

6. OCC-Optimized Collective Communication Library. `http://www.cs.utk.edu/pjesa/projects/occ/`

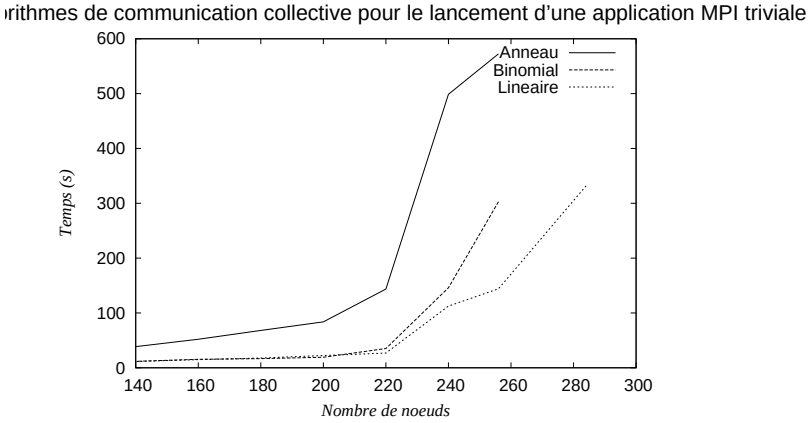


FIGURE *3.10 – Comparaison des temps de lancement de huit processus MPI par machine pour plusieurs algorithmes de Allgather.*

L'algorithme de Bruck présente les meilleures performances. Cependant, les performances de l'algorithme basé sur la composition de deux opérations collectives sur des arbres binomiaux sont relativement proches de celles de l'algorithme de Bruck. Ces deux algorithmes envoient le même nombre de messages ($2log_2(N)$). L'algorithme de Bruck effectue toutes ses communications *en phase*, c'est-à-dire qu'à chaque étape de l'algorithme un processus effectue un échange de message avec deux de ses voisins. Il effectue alors $log_2(N)$ étapes de communications. À l'inverse, l'algorithme binomial n'implique pas une participation égale de tous les processus. Il est effectué en deux phases, la première consistant à effectuer une concaténation vers un processus désigné comme racine ($log_2(N)$ étapes pour $log_2(N)$ messages), puis le tampon de données résultant est diffusé par ce processus racine auprès des autres processus ($log_2(N)$ étapes pour $log_2(N)$ messages). L'algorithme de Bruck effectue donc un nombre de phases inférieur, cependant il envoie un plus grand nombre de messages simultanés : il charge donc le réseau de façon plus importante que l'algorithme binomial, ce qui explique pourquoi la différence de performances entre ces deux algorithmes est relativement faible.

La figure 3.10 présente une comparaison des temps de lancement d'une application MPI pour plusieurs algorithmes de communications collectives dans ORTE1. L'application utilise ici huit processus par machine afin d'impliquer des tailles de messages plus importantes qu'avec un seul processus par machine, et ainsi rendre l'opération collective de allgather plus significative, tout en ne surchargeant pas de manière exagérée le nombre de processus MPI à gérer localement pour chaque démon de l'environnement d'exécution.

Cette mesure a été effectuée dans ORTE1 : on constate une forte augmentation du temps de déploiement à partir d'un certain nombre de processus. Ce saut est dû à la tempête de connexions entrantes vers le processus mpiexec, comme expliqué dans la section 3.2.2. On constate cependant un effet prononcé de l'utilisation de l'algorithme en anneau double, qui présente des temps de déploiement très supérieurs à ceux des autres algorithmes. Il est relativement surprenant de constater que l'algorithme présentant les meilleures performances est l'algorithme linéaire, qui implique que les processus envoient tous, un par un, leurs informations de connexion au processus mpiexec, puis que celui-ci leur envoie le résultat selon le même procédé.

Dans ORTE1, des connexions point-à-point directes peuvent être ouvertes à la demande entre deux démons de l'environnement d'exécution. L'arbre binomial nécessite d'ouvrir des connexions supplémentaires par rapport à celles déjà établies lorsque le démon rejoint l'environnement d'exécution. Ces connexions déjà établies sont utilisées par l'algorithme linéaire, qui n'a pas à en établir de nouvelle. À cette échelle, les communications selon un arbre binomial ne présentent pas des performances suffisamment meilleures que l'algorithme linéaire pour compenser le temps nécessaire à l'établissement des connexions.

L'approche présentée dans la section 3.2.2 présente donc l'avantage significatif d'établir ces connexions dès la construction de l'infrastructure de communications. La première communication collective n'a donc pas à payer le coût de l'étalissement de ces connexions.

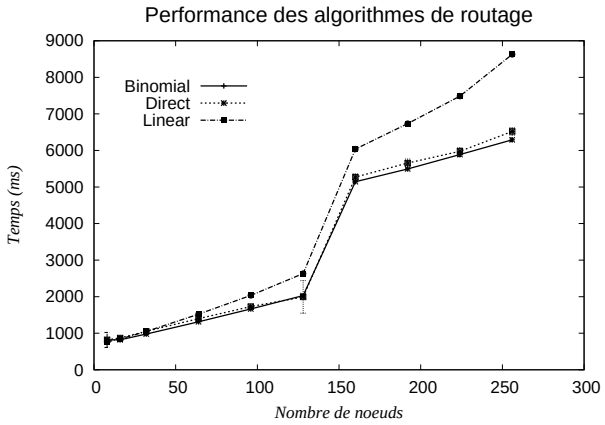


FIGURE *3.11 – Performances du lancement et de l'exécution de hostname sur une grappe pour différents algorithmes de routage dans ORTE2.*

3.4.2 Routage des communications

Comme exposé précédemment, ORTE2 permet de définir une topologie de routage qui est suivie par tous les messages transitant dans son infrastructure de communications. Si l'infrastructure de communications est construite directement le long de cet arbre de routage, les premières communications n'auront pas à établir celles-ci, comme on a vu dans le cas des communications selon un arbre binomial dans ORTE1.

Les communications hors-bande de l'environnement d'exécution doivent suivre cette topologie de routage. Par exemple, les entrées-sorties de l'application doivent remonter des processus MPI vers le processus mpiexec dans le cas de sorties, ou être distribués du processus mpiexec vers les processus MPI dans le cas d'entrées. La topologie de routage doit alors offrir de bonnes performances en tant que réseau de concentration (plsuieurs processus vers un) et également comme réseau de distribution (un processus vers plusieurs).

La figure 3.11 présente une comparaison des trois algorithmes de routage implémentés dans ORTE2, en déployant l'application et en construisant la topologie comme décrit dans la section 3.2.2. Nous avons utilisé l'application *hostname* comme banc de test : cette application effectue uniquement un affichage sur la sortie standard, qui doit être redirigée vers le processus mpiexec.

Les trois topologies de routage comparées fonctionnent comme suit. *Direct* consiste à établir une connexion point-à-point entre le processus mpiexec et tous les démons de l'environnement d'exécution. *Binomial* suit un arbre binomial dont la racine est le processus mpiexec. Enfin, *linear* construit une chaine entre les processus de l'environnement d'exécution : le processus n est connecté avec le processus $(n-1)$ et le processus $(n+1)$.

Afin de n'évaluer que les performances de l'infrastructure de communications, nous avons utilisé un déploiement à plat, et non pas selon l'arbre de routage. On constate sans surprise que la topologie chaînée du routage linéaire offre des performances en-deçà de celles des autres algorithmes de routage. La topologie binomiale présente des performances légèrement meilleures que la topologie directe, en forme d'étoile autour de processus mpiexec.

Cependant, comme le déploiement n'a pas été effectué selon l'arbre de routage, les topologies n'impliquant pas une connexion directe au processus mpiexec sont désavantagées par rapport à la topologie directe : en effet, la première communication est ralentie par l'établissement des connexions de l'infrastructure de communications.

La topologie en arbre binomial présente donc les meilleures performances, malgré ce handicap. Dans le cas où les processus MPI effectuent des opérations de sortie, ces opérations sont envoyées au processus mpiexec par des messages hors-bande. Dans l'application utilisée ici, tous les processus MPI effectuent un affichage au même moment : le processus mpiexec doit alors gérer des messages arrivant de tous les processus. L'utilisation d'une topologie en arbre permet de réduire le nombre de connexions entrantes vers le processus mpiexec, améliorant ainsi les performances malgré la nécessité d'effectuer des sauts de plus pour acheminer les messages vers leur destination.

3.5 Scalabilité par nœud

Chaque démon de l'environnement d'exécution a pour tâche de gérer les processus MPI s'exécutant localement sur sa machine. Actuellement les architectures tendant à utiliser un nombre croissant de cœurs sur les machines, il est alors nécessaire pour l'environnement d'exécution de présenter également une scalabilité *locale*, c'est-à-dire de pouvoir supporter plusieurs processus locaux tout en conservant de bonnes performances.

ORTE2 fonctionne de manière hiérarchique : les communications hors-bande sont effectuées entre les démons de l'environnement d'exécution qui, si besoin, mettent à contribution leurs processus locaux [7]. Par exemple, l'opération collective allgather effectuée lors du déploiement d'une application par ORTE2 se fait de la façon suivante :

- les processus MPI envoient leurs données à leur démon local ;
- une fois qu'il a les donnés de tous ses processus locaux, le démon effectue l'opération collective ;
- une fois l'opération collective terminée, le démon met le tampon résultant à la disposition de ses processus locaux.

La conséquence de cette hiérarchisation est une diminution du nombre de communications entre les machines, qui est alors fonction du nombre de machines impliquées mais pas du nombre de processus MPI.

Cependant, les quantités de données à transmettre sont fonction du nombre de processus MPI. Le nombre de processus par machine a donc une importance dans les temps de transmission des messages. De plus, ce fonctionnement hiérarchique implique des synchonisations locales entre les processus MPI au moment des communications collectives. De plus, le démon de l'environnement d'exécution doit lancer un par un tous les processus MPI exécutés sur sa machine, et donc faire autant de *fork()* successifs.

La figure 3.12 présente une étude de la scalabilité locale d'ORTE2. Toujours sur la grappe GdX, nous avons utilisé un nombre fixe de machines (huit) et augmenté le nombre de processus s'exécutant sur ces machines.

On constate que pour des applications non-MPI telles que *hostname* et */bin/true*, le temps de lancement est une fonction affine du nombre de processus par machine avec une epnte très faible. En effet, le déploiment de ces applications ne nécessite pas d'intialiser la bibliothèque MPI : le seul surcoût à payer correspond alors au nombre d'appels à *fork()* pour lancer les processus de l'application.

À l'inverse, le fait d'utiliser plusieurs processus par nœud a un impact significatif sur les performances de l'initialisation d'une application MPI. La synchronisation évoquée ci-avant n'existe pas pour un seul processus par machine, et provoque un ralentissement significatif du lancement dès que plus d'un processus est utilisé. Ce ralentissement est alors particulièrement visible lorsque l'application passe d'un à deux processus par machine. Pour un plus grand nombre de processus, le temps de lancement croît ici encore selon une fonction affine donc la pente est plus marquée que pour les applications non-MPI : la synchronisation locale nécessitée par les opérations collectives induites par l'initialisation de la bibliothèque MPI, et l'augmentation des volumes de données échangés lors des échanges d'informations entre les processus ont un impact sur le temps de déploiment de l'application.

3.6 Conclusion

Ce chapitre présente des travaux relatifs au passage à l'échelle des mécanismes des environnements d'exécution, notamment le déploiement de l'environnement d'exécution, la mise en place de l'infrastructure de communications, le lancement et l'initialisation de l'application et le transfert des communications hors-bande. Il est montré le rôle central de l'infrastructure de communications dans le fonctionnement de l'environnement d'exécution et dans les services rendus à l'application.

On a vu l'importance de la topologie de déploiement de l'environnement d'exécution, ainsi que celle des communications collectives et du routage. L'idée principale de ce chapitre consiste à utiliser la première pour construire les autres : l'infrastructure de communications est construite au fur et à mesure de la mise en place de l'environnement d'exécution, le long de son arbre de déploiement. La topologie de l'infrastructure de communications correspond à celle imposée par l'algorithme de routage, afin d'utiliser les connexions déjà établies lors de la construction de l'infrastructure de communications. Le fait de construire celle-ci en suivant

7. ORTE1 proposait un mode *direct*, où la couche ORTE des processus MPI permettait de communiquer directement avec ceux-ci. Ce mode a été abadonné lors du passage à ORTE2.

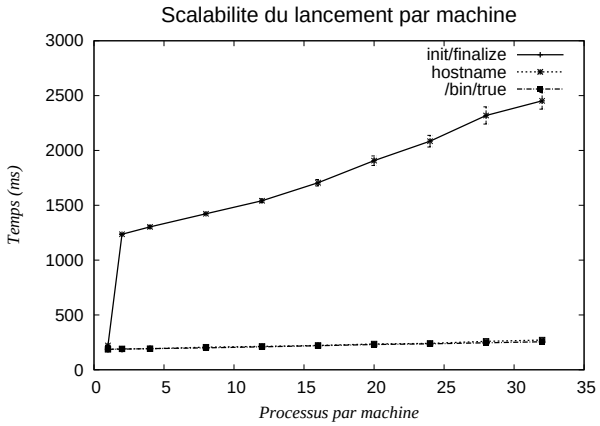


FIGURE *3.12 – Scalabilité du lancement de plusieurs processus par machine, sur huit machines.*

l'arbre de déploiement permet de le faire de façon distribuée et de ne pas créer de tempête de connexion sur un processus central.

La progression du lancement de l'application de manière asynchrone, sans attendre que tout le système ait terminé une étape pour passer à la suivante, permet également de tirer parti de l'unification de ces topologies en permettant de propager des commandes sur un arbre en cours de construction, et limiter les temps d'attente durant le déploiement d'une application à grande échelle.

Cette intégration plus serrée de la construction de l'infrastructure de communications avec le déploiement de l'environnement d'exécution va à l'encontre des pratiques actuelles d'interfaçage avec des systèmes de lancement externes : en effet, ceux-ci ne permettent pas de modifier la commande à lancer au cours du déploiement lui-même. L'intégration des mécanismes présentés ici nécessite donc une collaboration plus fine entre le système de lancement et l'environnement d'exécution qui n'est pas disponible actuellement.

Enfin, on a vu ici des topologies de communications simples, construites en une seule étape (lors du déploiement de l'environnement d'exéuction). Des topologies comme le graphe binomial [126, 6] présentent des caractéristiques les rendant particulièrement adaptées aux environnements d'exécution et aux mécanismes mis en œuvre dans ceux-ci. Par exemple, la topologie du BMG permet d'utiliser ses liens directs pour effectuer des communications collectives en utilisant des algorithmes qui ont été démontrés comme optimaux en termes de nombre de messages et de nombre d'étapes, comme l'arbre binomial pour la distribution ou l'algorithme de Bruck [34] pour la concaténation avec redistribution du résultat.

Une fois lancée, l'application doit pouvoir poursuivre son exécution jusqu'à la fin. La grande échelle pose un problème de fiabilité : la fréquence des pannes survenant dans le système nécessite de pouvoir maintenir l'exécution malgré ces pannes. Le chapitre suivant traite du support apporté par l'environnement d'exécution aux mécanismes mis en œuvre auprès des applications pour tolérer les défaillances.

Chapitre 4

Tolérance aux pannes

Même avec les composants les plus stables, des pannes risquent de survenir dans les systèmes à grande échelle. Du fait de l'addition des probabilités de pannes des composants d'un système, chaque composant ayant une probabilité donnée de tomber en panne, si l'on augmente le nombre de composants dans ce système, la probabilité qu'une panne survienne dans ce système augmente [145]. Une machine de l'ensemble sur lequel l'application s'exécute est alors perdue pour ce calcul.

Le comportement communément admis des systèmes de calcul à hautes performances en cas de défaillance, et toléré par la norme MPI dans ses versions 1.x et 2.x, consiste à terminer l'application en cas de panne. Une panne dans un composant du système, si elle n'est pas tolérée, fait perdre le calcul exécuté par le système dans son ensemble. Les calculs exécutés sur de tels systèmes étant souvent destinés à durer plusieurs heures voire plusieurs jours, la probabilité qu'une panne survienne pendant le calcul est forte et la perte engendrée importante.

Dans l'hypothèse où les pannes sont indépendantes les unes des autres et d'après la formule des probabilités disjointes, la distribution de probabilité qu'un composant d'un système constitué de n composants tombe en panne est égale à la somme des distributions de probabilités de pannes de ses composants.

Si des probabilités sont disjointes, c'est-à-dire si les pannes sont indépendantes les unes des autres, le temps moyen avant défaillance d'un composant est égal à son temps médian avant défaillance [1]. On obtient alors le temps moyen avant défaillance du système, noté $MTBF$, grâce à la formule donnée par l'équation 4.1 [145] ($MTBF_i$ désignant le temps moyen avant défaillance du composant i). On peut définir le temps médian (ou moyen) avant défaillance comme le temps d'exécution d'une application ayant une chance sur deux de se terminer.

$$MTBF_{total} = (\sum_{i=0}^{n-1} \frac{1}{MTBF_i})^{-1} \tag{4.1}$$

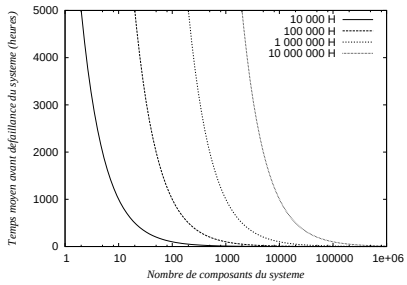


FIGURE *4.1 – Temps moyen avant défaillance pour des systèmes utilisant des composants de différents temps moyen avant défaillance individuels.*

Intuitivement, on voit que la fiabilité d'un système est minorée par la fiabilité de composant le moins fiable en faisant partie. De plus, on voit grâce à l'équation 4.1 que, même si la probabilité de panne durant un intervalle de temps donné d'un composant est faible, la probabilité de panne d'un seul composant d'un système à grande échelle est élevée.

En utilisant la formule donnée par l'équation 4.1, la figure 4.1 montre le temps moyen avant défaillance d'un système en fonction de sa taille pour des composants de différents temps moyens avant défaillance. Le temps moyen avant défaillance étant une fonction hyperbolique de la taille du système, même pour des composants arbitrairement fiables, le temps moyen avant défaillance d'un système à grande échelle est de quelques heures seulement. Par exemple pour un système dont les composants ont un temps moyen avant défaillance individuel de 10 000 heures, un système comptant 1 000 composants

1. On considère généralement que la loi de probabilité de défaillances est une loi symétrique d'espérance mathématique inverse à la fiabilité de ce composant, et donc son espérance, sa médiane et sa moyenne sont égales.

aura un temps moyen avant défaillance de 10 heures. En considérant des composants particulièrement fiables, ayant un temps moyen avant défaillance de 100 000 heures, un système constitué de 10 000 composants aura un temps moyen avant défaillance global de dix heures et un système constitué de 100 000 composants aura un temps moyen avant défaillance global de une heure seulement. Par comparaison, les machines les plus puissantes actuellement sont constituées de plusieurs centaines de milliers de composants [2].

Si aucun mécanisme de tolérance aux pannes n'est mis en place dans le système, tout le calcul en cours est perdu lorsqu'une panne survient. Il est donc dans ces conditions indispensable de pouvoir tolérer les fautes de manière à ce que le calcul puisse continuer à s'exécuter et se terminer malgré les pannes, tout en conservant des performances maximales (objectif de hautes performances). Il est donc indispensable de pouvoir tolérer les pannes dans les systèmes à grande échelle, où les pannes surviennent fréquemment.

L'environnement d'exécution d'une applications MPI doit donc surveiller l'état de l'application, détecter les défaillances et déterminer et coordonner les actions à prendre pour réagir à la défaillance. Dans le cas où celles-ci ne sont pas tolérées, il propage la terminaison de l'application. Si un mécanisme spécifique est mis en place pour les tolérer, ce mécanisme est supporté par l'environnement d'exécution en utilisant des techniques qui seront détaillées dans ce chapitre, avec une implication plus ou moins forte de l'environnement d'exécution dans la tolérance aux défaillances. L'environnement d'exécution détecte les défaillances grâce au système de détection de fautes, et les corrige en utilisant un mécanisme implémenté dans des composants spécialisés.

L'objectif suivi dans ce travail est de tolérer les pannes dans les applications parallèles par passage de messages de manière transparente. Cette transparence s'exprime par le fait que le programme n'a pas à être modifié pour la tolérance aux pannes : un code développé dans un environnement non tolérant aux pannes pourra s'exécuter dans un environnement tolérant aux pannes, et inversement. La tolérance aux pannes n'a pas à être prise en charge par l'application. Du point de vue de l'utilisateur, il peut avoir à soumettre à nouveau l'exécution dans le système de réservation, ou ne s'apercevoir de rien.

Les défaillances étant un problème inhérent aux systèmes à grande échelle, il faut que les mécanismes utilisés pour les tolérer soient capables de passer à l'échelle.

4.1 Modèles et définitions

4.1.1 Modèle pour les systèmes distribués

On considère ici un système distribué comme un ensemble de processus communiquant entre eux par des canaux de communications.

Un processus est caractérisé par son état à un instant donné. Au cours d'une exécution, il passe par une succession d'états pour aller de l'état initial à l'état final. Les changements d'états se font par le moyen d'évènements déterministes on non-déterministes.

Un canal de communications est un médium utilisé pour transmettre des informations entre un émetteur et un receveur. On considère ici des systèmes asynchrones : tout message émis dans un canal de communications est reçu après un temps fini mais arbitrairement long.

L'état d'un système distribué est donc définissable par l'ensemble des états des processus le constituant ainsi que les messages en transit dans les canaux de communications.

4.1.2 Point de reprise d'un processus

L'enregistrement d'un point de reprise d'un processus consiste à enregistrer son état de manière à rendre possible un redémarrage dans cet état à partir des données enregistrées. Ces données sont écrites dans un fichier qui peut être transféré sur une autre machine pour reprendre l'exécution du processus dans l'état dans lequel il a été enregistré.

À chaque processus sont associés :
- un programme qui détermine ce que doit faire le processus
- une entrée et une sortie
- un état représenté par la valeur des variables du programme
- un compteur de programme et des registres

2. Au classement du Top500 de juin 2009, six supercalculateurs étaient composés de plus de 100 000 cœurs.

– une pile contenant les appels des fonctions du programme
– des ressources allouées à l'application
– un contexte d'exécution de l'application
– des bibliothèques chargées, des fichiers ouverts

Les bibliothèques utilisées présentent des limitations techniques sur ce qui peut être écrit dans le point de reprise. Par exemple, BLCR [107, 148] peut sauvegarder une région de mémoire mappée propre à un processus, mais ne peut sauvegarder une région partagée que sous certaines conditions (la zone de mémoire doit avoir été obtenue avec `mmap()`). Les mécanismes de communications inter-processus comme les sémaphores ne peuvent pas être sauvegardés car il s'agit d'éléments de communications.

Dans la limite des éléments qui peuvent être enregistrés, la prise d'état d'un processus consiste à enregistrer tout ce qui peut l'être dans un point de reprise.

4.1.3 Types de fautes considérées

Une revue des types de fautes et des approches traditionnelles pour y faire face peut être trouvée dans le premier chapitre de la thèse de Denis Conan (datant cependant de 1996) [56]. Une faute y est caractérisée par quatre attributs : sa nature (intentionnelle ou accidentelle), sa durée (temporaire, permanente ou définitive), son origine (due à un phénomène physique ou humain, provoquée par une partie de l'état du système ou par son environnement, due à la conception ou à l'exploitation de l'application...) et son étendue (portion du système affectée).

Avižienus *et al.* [14, 13] proposent une classification faisant une distinction entre les termes faute, défaillance et erreur : une *erreur* est une partie fausse de l'état du système, qui peut être latente (ou *dormante*) tant que le système n'utilise pas cette partie fausse ; une *faute* en est la conséquence ; une *vulnérabilité* est une faute interne qui permet à une faute externe d'atteindre le système ; une *défaillance* correspond à l'incapacité du système ou d'un de ses composants à assurer le service qui lui est demandé ; une *faute* est caractérisée comme ci-avant, par sa nature, sa durée, son origine et son étendue.

Selon une approche pragmatique, on peut ne s'intéresser qu'à l'effet de la faute sur le système : son influence sur le système et sur le fait qu'une application pourra ou non continuer son exécution dans un certain contexte dépend de la façon dont la faute l'affecte et non pas des autres caractéristiques. Une machine qui s'arrête affectera le système de la même façon que cette panne soit causée par une surchauffe du processeur, un bug du système d'exploitation ou un technicien qui se prend les pieds dans un câble d'alimentation.

On peut alors synthétiser les études citées ci-avant en classant les fautes selon trois grandes catégories : les pannes temporaires, les fautes byzantines (qui peuvent être également considérées comme des erreurs) et les pannes franches.

Les pannes temporaires peuvent être de plusieurs natures différentes. On parle d'omission lorsqu'un processus ou un canal ne réalise pas une action. À l'inverse, la duplication correspond au fait qu'un canal ou un processus réalise deux fois une action qui n'aurait dû se produire qu'une seule fois. Enfin, une inversion correspond à la réalisation d'une action B avant une action A, alors que A aurait dû avoir lieu avant B.

Les fautes byzantines peuvent être temporaires ou définitives. Un processus ou un canal a un comportement arbitraire erroné, mais la détection des erreurs ne se fait pas aussi facilement que dans les cas dans autres types de fautes. Ainsi, l'entité fautive peut interagir avec les autres entités du système, et la faute se propage dans le système.

Enfin, les pannes franches sont des pannes définitives : un processus ou un canal ne réalise plus aucune action. On parle alors de silence sur défaillance. Comme défini dans [140], ces pannes sont des omissions permanentes. On s'intéresse ici à ce type de pannes, appelées aussi *pannes crash* ou *fail stop*. Une caractérisation des pannes franches peut être trouvée dans [149], avec le mode *arrêt sur défaillance*, qui est une extension du mode *silence sur défaillance* qui implique que tous les composants du système aient connaissance de la panne.

Les défaillances considérées dans le cadre de cette étude sont les pannes franches. Une panne franche peut correspondre à une défaillance matérielle : panne franche du processeur, coupure physique (électricité, réseau), ou à une erreur logicielle (désordonnancement accidentel causé par un bug du l'ordonnanceur, bug logiciel local, etc).

Composants fiables La probabilité qu'une panne touche un système augmentant avec le nombre de composants de ce système, on fait l'hypothèse qu'un sous-ensemble du système considéré composé d'un petit nombre de composants est *fiable*. En pratique, on considère comme fiable un composant dont le temps moyen avant défaillance est grand devant le temps d'exécution typique d'une application sur le système dont il fait partie. Certaines approches de la tolérance aux pannes nécessitent le fait que certains composants du système ne seront pas concernés par des pannes au cours de l'exécution d'une application. Ces composants fiables étant en nombre limité, on fait alors l'hypothèse (forte) que ces composants ne seront pas victimes d'une défaillance. Les pannes n'étant pas plus probables sur les machines composant les grappes à grande échelle que sur les machines actuelles, cette hypothèse est réaliste.

4.2 Environnements d'exécution tolérants aux pannes

Les applications parallèles et distribuées tolérantes aux pannes doivent être supportées par un environnement d'exécution tolérant aux pannes. Selon le protocole utilisé au niveau de l'application, ce support peut impliquer un plus ou moins grand nombre de composants durant l'exécution, et le comportement à suivre en cas de panne nécessitera une implication plus ou moins importante de l'environnement d'exécution.

4.2.1 Pour MPI

Des implémentations tolérantes aux pannes de la norme MPI permettent à l'application de poursuivre son exécution malgré les pannes. Le rôle de l'environnement d'exécution dans la tolérance aux pannes dépend alors du protocole utilisé. La section 4.3 présente deux grandes catégories de protocoles de retour sur points de reprise : coordonnés ou non coordonnés.

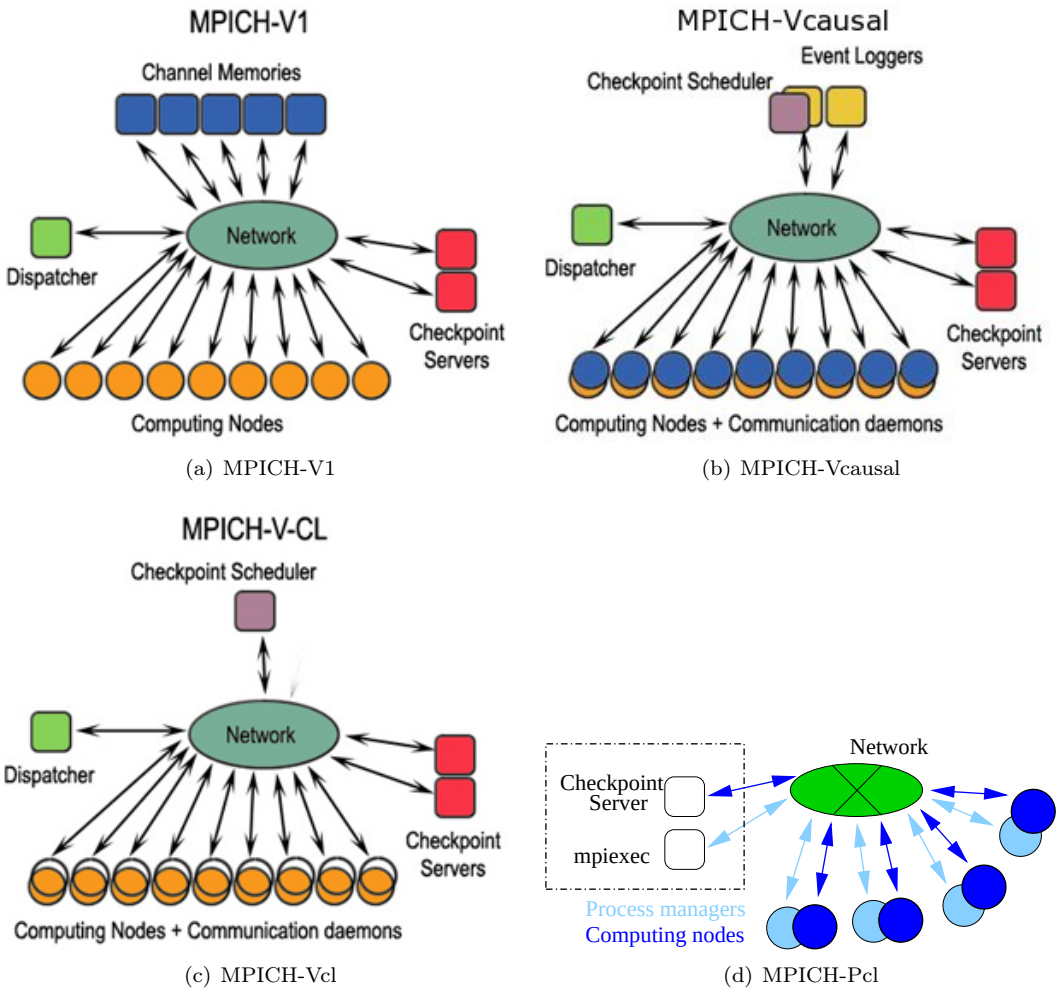
Dans le premier cas, tous les processus effectuent une prise de point de reprise au même moment : en cas de panne, ils effectuent tous un retour en arrière. Les implémentations de ces protocoles dans LAM-MPI [148], OpenMPI [105, 104] et MPICH-V [31, 32, 119] tiennent compte de ce dernier point et terminent l'application en cas de panne : les processus survivants n'ont pas à continuer leur exécution, puisque l'application entière va effecteur un retour en arrière. Au redémarrage, l'environnement d'exécution peut être entièrement redéployé et réinitialisé, comme pour MPICH-V. Il doit alors avoir connaissance du fait qu'il s'agit d'un redémarrage, et obtenir des points de reprise des processus pour redémarrer l'application à partir d'un état précédemment enregistré, et non pas démarrer l'application elle-même. À l'inverse, le protocole de retour sur points de reprise coordonnés implémenté dans Charm++ [109] ne redéploie pas son environnement d'exécution. Celui-ci est au préalable déployé sur l'ensemble des nœuds potentiellement utilisés dans le calcul, y compris ceux qui seront utilisés pour remplacer des nœuds tombés en panne. Lorsqu'une panne survient, les processus survivants effectuent un retour en arrière à partir de leur point de reprise local, et un nœud disponible redémarre les processus perdus par la panne. Ainsi, l'environnement d'exécution n'a pas à être redéployé, mais il doit être capable de cicatriser lorsqu'un nœud disparaît.

L'environnement d'exécution doit être étendu pour supporter le mécanisme de tolérance aux fautes par retour sur points de reprise avec un composant additionnel : le serveur de points de reprise, exécuté sur un support supposé fiable. Il peut être rendu lui-même tolérant aux pannes grâce à des techniques de réplication afin de le rendre plus robuste [25].

Dans le cas des protocoles non-coordonnés, l'environnement d'exécution doit survivre à la panne pour supporter le rétablissement de l'état de l'application tout en permettant aux autres processus de l'application de continuer à s'exécuter normalement, en accord avec le protocole de recouvrement d'état au niveau applicatif. Il ne doit donc pas terminer l'application, mais cicatriser de façon à pouvoir relancer l'application.

L'environnement d'exécution de base de MPICH2, appelé MPD [39], est tolérant aux pannes dans le sens où il continue à fonctionner après une panne : sa topologie en anneau se reforme avec un processus de moins. MPD est partagé entre plusieurs applications MPI, et cet anneau ne supporte pas directement les applications : lorsqu'une application est lancée, les démons MPD situés sur les machines utilisées par cette application se clonent pour former l'instance de l'environnement d'exécution qui supportera cette application en particulier. Cette instance de l'environnement d'exécution n'est pas tolérante aux pannes.

Les implémentations des protocoles non coordonnés dans MPICH-V [21, 29, 31, 32, 119, 30] sont supportés par une extension de l'environnement de MPICH [96] afin de le rendre tolérant aux pannes. Tous les processus



(a) MPICH-V1 (b) MPICH-Vcausal

(c) MPICH-Vcl (d) MPICH-Pcl

FIGURE *4.2 – Architecture des protocoles implémentés par MPICH-V, et composants de l'extension de l'environnement d'exécution de MPICH.*

de l'environnement d'exécution sont connectés à un processus central, correspondant au point de lancement de l'application (*mpiexec*). En cas de panne, celui-ci détecte l'arrêt d'un processus de l'environnement d'exécution et le relance sur une nouvelle machine.

Les protocoles à points de reprise non coordonnés impliquent également des composants supplémentaires de l'environnement d'exécution afin de supporter la journalisation des messages ou l'enregistrement des relations de causalité entre les messages. Le protocole à mémoire de canal implémenté dans MPICH-V1 [21] nécessite la mise en place de composants stables pour enregistrer les messages envoyés entre les processus. Ces mémoires de canal sont des points d'engorgement pour les applications communiquant intensivement, et le nombre de processus par mémoire de canal doit être choisi afin de limiter cet engorgement. Les protocoles à journalisation de messages implémentés dans MPICH-V2 [29, 31, 32, 119] et MPICH-V*causal* [30] ne nécessitent pas de mémoire de canal pour enregistrer les messages, mais ils nécessitent un enregistreur d'évènements pour enregistrer les relations de causalité entre les messages.

Les extensions qui ont été faites à l'environnement d'exécution de MPICH et MPICH2 sont représentés figure 4.2.

La détection de fautes peut se faire en utilisant les propriétés des canaux de communications (timeout TCP, détection physique pour les types de réseaux utilisant un codage de canal permettant de détecter les déconnexions comme NRZ...), ou grâce à un mécanisme de détection de fautes [1] intégré à l'environnement

d'exécution. P2P-MPI utilise un détecteur de défaillances prévenant les processus MPI essayant de communiquer avec un processus que celui-ci a été victime d'une panne, indépendemment des problèmes de communications notifiés par le réseau [92].

4.2.2 Pour d'autres types d'applications

Le réseau de couches MRNet [10, 147] peut servir d'environnement d'exécution à des applications parallèles et distribuées tolérantes aux pannes en permettant de relancer un processus de son infrastructure qui a été victime d'une panne, et en lui permettant de retrouver son état [9], comme présenté plus en détails dans la section 4.3.2. Cependant, le relancement du processus MRNet victime de la panne et le relancement du processus de l'application ne sont pas pris en charge directement par MRNet. De plus, si une extension de l'environnement d'exécution doit être faite pour supporter un protocole de tolérance aux pannes au niveau de l'application, celle-ci doit être écrite spécifiquement : MRNet est plus une API pour écrire des environnements d'exécution qu'un environnement d'exécution en lui-même.

La bibliothèque d'algèbre linéaire tolérante aux pannes FT-LA [51, 24] a été conçue pour être utilisée avec FT-MPI [74]. La tolérance aux pannes est dans ce cas gérée explicitement par l'application : l'environnement d'exécution lui permet de mettre en place ses mécanismes et ne met en place aucun protocole de tolérance aux pannes par retour sur points de reprise. Des opérations de calcul matriciel itératives ajoutent une redondance dans les données à calculer à chaque étape d'un calcul itératif. Ces données supplémentaires, combinées aux données contenues dans les processus survivants, permettent de retrouver les données perdues dans la défaillance d'un processus, et de reprendre le calcul sans avoir à effectuer de retour en arrière. Le support de l'environnement d'exécution consiste à relancer un processus et à le positionner à la place du processus victime de la panne.

La technique des points de reprise sans disque [137, 138] consiste à enregistrer l'état des processus d'une application selon un protocole coordonné sans journalisation de messages et à conserver l'état des processus dans la mémoire volatile d'autres processus. En ce sens, l'enregistrement des points de reprise n'est pas effectué par l'environnement d'exécution mais par l'application elle-même. Une application de ce protocole est disponible pour PVM [154].

PVM permet de notifier à l'application un changement dans l'état de la machine virtuelle, en particulier lorsqu'un nœud est victime d'une panne. Condor [120] peut utiliser PVM comme son environnement d'exécution et tirer parti de ces notifications, par exemple avec FatCop [50]. Cette notification est fournie par HARNESS [75, 73], l'environnement d'exécution de PVM. HARNESS est un environnement d'exécutions *généraliste*, dans le sens où il n'est pas destiné à un type d'applications en particulier. Il supporte également FT-MPI [74]. Lorsqu'un nœud est victime d'une panne et disparaît, les autres nœuds sont notifiés de cette disparition. L'infrastructure de communications de HARNESS, basée sur des topologies présentant des propriétés de résilience comme le *k-ary sibling tree* [7], permet de cicatriser elle-même et de s'adapter à la disparition d'un nœud.

4.3 Tolérance aux pannes dans les systèmes distribués

Si l'enregistrement d'un point de reprise d'une application mono-processus en vue de le redémarrer après un arrêt est une opération simple comme on l'a vu dans la section 4.1.2, tolérer les pannes dans application distribuée n'est pas un problème trivial. Comme vu ci-avant, l'état d'une application distribuée est défini comme un ensemble d'entités pour lesquelles il n'existe pas de vision globale.

Il existe deux approches pour tolérer les pannes dans un système distribué en restaurant son état après la défaillance d'un de ses processus : le retour arrière sur points de reprises, qui consiste à rétablir un état du système qui a été atteint précédemment (*rollback recovery*), et la restauration de l'état sans retour en arrière (*forward recovery*).

La redondance est une solution relativement ancienne et qui est moins étudiée dans le domaine d'applications présent : en effet, si un taux de redondance de N permet de tolérer N défaillances dans une application, elle divise aussi les ressources de calcul utilisables pour faire avancer le calcul par N. Cette approche est encore utilisée pour certaines applications sur les grilles. L'attention ici se porte sur le support de protocoles au niveau applicatif sans redondance, utilisant des points de reprise, coordonnés ou non, ou des données déjà présentes dans le système pour restaurer l'état d'une application.

4.3.1 Retour sur point de reprise

Comme vu dans la section 4.1.1, l'exécution d'un processus est une succession d'actions qui le font passer d'un état à un autre. La reprise sur point consiste à faire reprendre son exécution à un processus à partir d'un état précédent sauvegardé lors d'une exécution précédente au moyen d'un point de reprise. La prise et la sauvegarde des points de reprise des processus d'une applications distribuée ne peut pas être effectuée de manière indépendante entre les processus. Pour assurer le maintien de la cohérence de l'état de l'application, les prises de points de reprise doivent être effectuées selon un protocole donné.

Les protocoles de retour sur points de reprise impliquent que les différents points de reprise soient sauvegardés sur un support de stockage que l'on considère comme stable afin de ne pas être concernés par une panne. Un processus situé sur une machine tombée en panne peut être migré sur une machine saine et reprendre son exécution à partir d'un point de reprise précédemment sauvegardé.

Il existe plusieurs catégories de protocoles, dont une revue peut être trouvée dans [72]. On peut aussi trouver un aperçu des implémentations existantes (en 2002) dans [98].

Les protocoles permettant à des applications de tolérer les pannes nécessitent un support de l'environnement d'exécutions. Nous allons voir dans la suite de cette section que ce support n'est pas le même selon le protocole utilisé par l'application. En particulier, les services rendus par l'environnement d'exécution au cours de l'exécution de l'application et lorsqu'une panne survient sont spécifiques à chaque protocole.

La section 4.3.1.1 définit la notion d'état cohérent pour une application distribuée et les conditions à remplir par un protocole de prise de points de reprise pour assurer l'exécution correcte d'une application. Les sections 4.3.1.2 et 4.3.1.3 présentent les principes généraux des deux grandes familles de protocoles de tolérance aux pannes par retour sur points de reprise ; enfin, la section 4.3.2 présente comment la tolérance aux pannes peut être effectuée sans points de reprise dans certaines conditions.

4.3.1.1 État cohérent et ligne de recouvrement

Pour pouvoir reprendre le calcul après une panne, on cherche dans l'approche suivie par les protocoles de retour sur points de reprise à assurer le fait que tous les processus atteignent un état cohérent après une panne et l'exécution du protocole de recouvrement d'état, c'est-à-dire d'un état dans lequel le système aurait pu être au cours du calcul et ne modifiant pas la séquence des évènements déterministes et le résultat d'un calcul déterministe si on revient dessus. Un état est dit *cohérent* si pour tous les messages m d'un processus P_i vers P_j, si le point de reprise sur P_j a été effectué après la réception de m alors le point de reprise de P_i a été effectué après l'émission de m [119].

Une application parallèle communiquant par passage de message est une application distribuée, et à un instant donné, on ne sait pas si un message est en train de transiter sur le réseau. Il faut donc tenir compte des messages qui circulent, et de l'absence d'horloge globale entre les processus ainsi que de vision globale du système. Si les processus effectuent leur prise de point de reprise indépendemment les uns des autres et sans tenir compte des messages circulant sur le réseau, le système risque de se retrouver dans un état incohérent après un retour en arrière d'un processus. Le processus qui est revenu en arrière peut par exemple attendre la réception d'un message provenant d'un autre processus qui, ayant déjà envoyé ce message, ne le renverra pas. Dans ce cas, on dit que des *processus orphelins* sont créés.

Lorsque le système ou un sous-ensemble de ses processus effectue un retour en arrière, il effectue une série d'actions jusqu'à se retrouver dans un état cohérent : on parle alors de *ligne de recouvrement*. Il faut évidemment que cette ligne de recouvrement soit la plus avancée possible, afin de perdre le moins possible de temps lors du retour en arrière et diminuer le coût de la faute. S'il est impossible actuellement de ne pas subir de surcoût lorsqu'une panne survient, la recherche dans ce domaine cherche à le diminuer le plus possible mais les fautes engendrent pour le moment inévitablement un surcoût en matière de temps d'exécution. Les protocoles de retour sur points de reprise coordonnés font repartir l'ensemble du système du dernier état enregistré : tous les processus effectuent un retour en arrière.

Les protocoles de retour sur point de reprise non coordonnés ne font revenir en arrière que le processus concerné par la panne, et peuvent faire revenir en arrière les processus dont l'exécution dépend causalement du processus qui est retourné en arrière : par exemple pour faire envoyer un message dont la réception s'est faite après la dernière prise de point de reprise. La propagation des retours en arrière des processus peut aller jusqu'à reprendre le calcul à partir du début, comportement appelé *effet domino* [143]. La tolérance aux pannes ne sert

alors à rien, si ce n'est à ralentir le calcul. L'enregistrement des messages échangés dans le but de pouvoir les rejouer en cas de retour en arrière permet d'éviter cet effet domino.

Dans tous les protocoles présentés ci-après, on suppose que les canaux de communication ont la propriété FIFO. En pratique, cette hypothèse dépend du canal de communication utilisé. De plus, le découpage des messages entre plusieurs canaux de communications pour multiplexer les communications lorsque plusieurs médias de communications sont disponibles ne garantit pas cette propriété : il faut descendre le protocole au niveau de chaque canal, et non pas au niveau des couches de communications de haut niveau, avant que le multiplexage soit effectué.

4.3.1.2 Protocoles coordonnés

L'algorithme de Chandy-Lamport est un exemple de prise de point de reprise de tout le système dans un état cohérent avec l'introduction de l'idée de instantané distribué [49]. Un instantané est formé par l'ensemble des points de reprise du système. Un processus du système a un rôle d'initiateur du point de reprise, en faisant circuler un marqueur. La prise de points de reprise est alors effectuée par une vague : chaque processus ayant reçu le marqueur le transmet à leur tour. La façon dont les messages sont traités pendant la vague de prise des points de reprise dépend de l'implémentation : ils peuvent être retardés pour être envoyés après la fin de la vague, ou enregistrés.

Points de reprise bloquants ou non bloquants Les protocoles de points de reprise bloquants arrêtent leur exécution pendant la prise du point de reprise et ne la reprennent que lorsque tous les processus l'ont fait. Ainsi il est impossible qu'un message ne traverse la vague de points de reprise.

Un processus du système joue le rôle d'initiateur de la vague de points de reprise : il envoie un marqueur à tous les autres processus du système. Chaque processus, lorsqu'il reçoit ce marqueur, arrête son exécution, envoie un marqueur à tous les autres processus du système (on parle de *requête de point de reprise*) et effectue sa prise de point de reprise. Les processus ne reprennent leur exécution qu'une fois qu'ils ont reçu les marqueurs de tous les autres processus. En supposant que les canaux de communications ont la propriété FIFO, il est impossible qu'un message double le marqueur ou inversement, et l'état global sauvegardé forme une coupe cohérente.

Les protocoles de point de reprise non bloquants n'arrêtent pas leur exécution au moment de la prise du point de reprise mais enregistrent les messages qui circulent entre le moment où ils prennent leur point de reprise et celui où ils ont reçu les marqueurs de tous les autres processus. Tous les messages reçus après la réception du marqueur et avant la réception d'un marqueur en provenance de l'expéditeur de ce message sont sauvegardés dans le point de reprise du processus.

Support de l'environnement d'exécution Les points de reprise doivent être stockés par l'environnement d'exécution sur un support fiable, comme défini dans la section 4.1.3. En cas de panne, l'environnement d'exécution peut finaliser l'application et la redémarrer à partir de la dernière vague de points de reprise sauvegardée ou s'auto-cicatriser, signaler aux processus survivants d'effectuer un retour en arrière sur leur point de reprise et redémarrer le processus victime de la panne à partir d'un point de reprise. Le déclenchement des vagues de points de reprise peut être effectué par l'environnement d'exécution en utilisant un ordonnanceur de points de reprise.

Avantages et inconvénients Après une panne, tous les processus du système redémarrent de l'état global précédent : il est impossible de rencontrer l'effet domino dans les protocoles de points de reprise coordonnés. De plus, il n'y a pas besoin de conserver en mémoire plus de deux images globales du système, réduisant l'espace de stockage nécessaire par rapport à d'autres protocoles et simplifiant le mécanisme de nettoyage des points de reprise inutiles.

Cependant, ces protocoles impliquent une synchronisation entre les processus, plus ou moins importante au moment de la vague de points de reprise selon l'implémentation, mais surtout au moment du rétablissement de l'état après une panne : tous les processus, même ceux qui n'ont pas été touchés par la panne, doivent effectuer un retour en arrière.

Une implémentation bloquante de l'algorithme de Chandy-Lamport est présentée section 4.4 et comparée avec l'implémentation non-bloquante de MPICH-Vcl.

4.3.1.3 Protocoles non-coordonnés

On peut ne faire repartir de l'état initial qu'un seul processus, et faire rejouer tous les messages. De cette façon, dans un premier temps seul le processus concerné par la panne repart du début de l'exécution, et peut ensuite forcer d'autres processus à revenir en arrière eux aussi pour lui envoyer un message. En apparence, le ralentissement de l'exécution lors d'une panne est alors moins important. Cependant il existe souvent des points de synchronisation dans les programmes par passage de message (barrières, transmissions de messages bloquantes), et le processus qui a effectué un retour en arrière ralentit forcément les autres. L'enregistrement de points de reprise intermédiaires permet alors de réduire l'amplitude des retours en arrière.

Un protocole où un processus peut forcer d'autres processus à revenir en arrière risque de conduire à l'effet domino, présenté ci-avant. Afin de l'éviter on peut enregistrer les messages envoyés entre les processus et les faire rejouer lors d'une rééxécution après une panne au lieu de faire revenir complètement des processus en arrière.

À l'inverse des protocoles de points de reprise coordonnés, les protocoles non coordonnés ne nécessitent pas de synchronisation des processus, ni au moment de la prise des points de reprise, ni au moment d'une panne. De plus, ils permettent plus de flexibilité dans l'ordonnancement des prises de points de reprise : chaque processus peut effectuer une prise de point de reprise localement à un moment qui, par exemple, optimiserait le nettoyage des messages sauvegardés localement, ou minimiserait son empreinte mémoire et donc la taille de son point de reprise. Cependant, ils nécessitent de sauvegarder les messages échangés ainsi que des informations sur l'ordre des messages, ce qui introduit un surcoût au cours de l'exécution.

La *piecewise deterministic asumption (PWD)* Un programme est dit déterministe si, à partir d'un état initial donné, il arrive toujours au même état final.

Un processus faisant partie d'un système réparti communiquant avec les autres processus par passage de messages a une exécution constituée d'une succession d'évènements non-déterministes entre lesquels l'exécution est déterministe. Dans une application MPI, les évènements non-déterministes sont par hypothèse les réceptions de messages. En cas d'utilisation d'un générateur de nombres aléatoires, la graine de celui-ci sera sauvegardée dans le point de reprise des processus ; un nombre aléatoire généré après la prise d'un point de reprise sera le même lors de la rééxécution du processus à partir de ce point de reprise. Les entrées-sorties sont des évènements non-déterministes ; on peut choisir d'effectuer une prise de point de reprise à chaque fois qu'une opération d'entrée-sortie est faite par un processus, ou introduire comme limitation le fait de ne pas pouvoir faire ce type d'opérations.

D'après cette hypothèse sur les évènements non-déterministes, la séquence d'évènements faisant passer un processus d'un état à un autre entre deux réceptions de messages est déterministe. Ainsi, à partir de l'état d'un processus juste après la réception d'un message, on arrive toujours au même état au moment de la réception du message suivant.

L'hypothèse selon laquelle si l'on rejoue les mêmes évènements non-déterministes, alors l'état final du système sera toujours le même pour un état initial donné s'appelle la *piecewise deterministic asumption* ou PWD [152].

Journalisation de messages Les protocoles de journalisation de messages sont des protocoles de prise de points de reprise non-coordonnés : chaque processus effectue ses points de reprise indépendamment des autres processus. Au cours de l'exécution, chacun enregistre les messages échangés avec les processus afin de pouvoir effectuer à nouveau les réceptions en cas de retour en arrière en s'appuyant sur la PWD. L'ordre causal des évènements est enregistré afin de pouvoir les rejouer dans le même ordre que lors de l'exécution initiale.

Les messages sont identifiés de manière unique par l'émetteur comme par le receveur grâce à une horloge logique (scalaire) locale, formant un *déterminant* contenant les horloges logiques de l'émetteur et du récepteur du message. À chaque évènement (émission ou réception), le processus incrémente une horloge locale. La valeur de cette horloge permet de savoir à quel moment un processus a reçu ou envoyé un message donné dans la succession d'évènements qui sont survenus pour ce processus. L'ensemble des déterminants permet de reconstituer l'ensemble des évènements qui sont survenus dans le système au cours d'une exécution.

Un déterminant est constitué de cinq entiers :

$$[id\_source, id\_dest, H_s, H_d, type] \tag{4.2}$$

Dans le déterminant, *id_source* représente le rang de l'expéditeur du message, H_s son horloge logique au moment de l'envoi du message, *id_dest* le rang du destinataire, H_r son horloge logique au moment de la réception du message et *type* représente le type d'évènement dont on enregistre l'identifiant de causalité pour le cas où cet évènement ne serait pas une émission ou une réception de message.

Les déterminants sont sauvegardés sur un élément de l'environnement d'exécution supposé fiable et appelé *enregistreur d'évènements*.

En cas de panne, le processus concerné effectue un retour en arrière et reprend son exécution au moment du dernier point de reprise. Il obtient auprès de l'enregistreur d'évènements et des autres noeuds les informations de causalité concernant les messages qu'il doit demander à rejouer. Il a alors connaissance de l'ordre dans lequel ils doivent être rejoués grâce aux déterminants. Ainsi, en vertu de la PWD, on se retrouve retrouve le même segment d'exécution qu'avant la panne. Par ailleurs, les protocoles doivent sauvegarder tous les évènements non-déterministes ayant une influence sur le système. Il est possible que des évènements n'ayant pas d'influence sur le reste du système ne soient pas enregistrés. Ainsi le segment d'exécution peut diverger de celui qu'il aurait été en l'absence de panne.

Support de l'environnement d'exécution L'environnement d'exécution doit permettre de sauvegarder les points de reprise sur un support fiable, comme défini dans la section 4.1.3. Les points de reprise étant pris de manière non-coordonnée entre les processus, un ordonnanceur de points de reprise permet de mettre en place des politiques de prise de points de reprise optimisant par exemple l'espace mémoire utilisé par le point de reprise ou le nombre de messages à conserver en mémoire dans les autres processus. Un enregistreur d'évènements permet de conserver la liste des déterminants construits par chaque processus et doit également être un composant fiable. Enfin, l'environnement d'exécution doit survivre aux défaillances. Lorsque des processus sont victimes d'une panne, il doit continuer à offrir son service aux processus survivants tout en supportant le redémarrage des processus victimes de la panne. Il doit alors s'auto-cicatriser, redémarrer les processus victimes de la panne à partir de leur dernier point de reprise enregistré et lui fournir les informations nécessaires à la restauration de son état (en particulier la liste des déterminants pour les protocoles à journalisation de messages).

4.3.2 Restauration de l'état sans point de reprise

Il est parfois possible de restaurer l'état d'un système sans avoir à effectuer de retour en arrière, mais en utilisant des informations déjà présentes dans ce système. Les protocoles de restauration d'état sont plus spécifiques à l'application concernée que les protocoles de retour arrière sur points de reprise.

Il est possible de mettre en place une distribution des données redondantes dans les autre processus du système en maintenant un état cohérent de redondance. Cette cohérence peut être faible, dans le sens où plusieurs copies de chaque donnée existent dans le système avec différents niveaux de cohérence.

Cette approche est typiquement adaptée aux réseaux de couches. En cas de panne, de nouveaux processus prennent la place des processus victimes de la panne dans la topologie de communication et rétablissent leur état en obtenant ces données à partir de leurs voisins. La localité des données redondées dans les processus survivants assure une limitation du nombre de processus impliqués dans le rétablissement de la structure après une défaillance et un confinement de la défaillance en limitant ainsi le nombre de processus impactés.

Il est possible d'introduire des codes correcteurs d'erreur dans les applications, par exemple en ajoutant une redondance dans les données à calculer à chaque étape d'un calcul itératif. Ces données supplémentaires, combinées aux données contenues dans les processus survivants, permettent de retrouver les données perdues dans la défaillance d'un processus, et de reprendre le calcul sans avoir à effectuer de retour en arrière.

Si la réplication n'est plus très utilisée du fait de la division des ressources qu'elle impose, des applications où les pannes sont très fréquentes peuvent utiliser une stratégie basée sur la réplication passive. Un maître est élu parmi les instances d'un processus, et sert de point de contact pour le reste de l'application. Si un processus meurt, il est détecté par le reste de l'application s'il s'agit d'un maître. Dans ce cas, un nouveau maître est élu parmi les répliquas s'il en existe toujours, et il prend la place du processus mort dans l'application. Il n'est alors pas nécessaire de redémarrer de nouveau processus pour remplacer celui qui a été victime de la panne : dans ce cas, on diminue le niveau de tolérance aux pannes (on peut subir une panne de moins au niveau du processus maître).

Une approche différente de la tolérance aux pannes et dite "*proactive*" consiste à prévenir la défaillance et à agir avant qu'elle ne survienne. En se basant sur l'observation de la machine et de son état (par exemple, sa température) des processus peuvent être migrées sur des machines saines avant qu'une panne ne risque de survenir.

4.3.3 Comparaison des deux approches

Le rétablissement de l'état du processus victime de la panne par retour arrière sur points de reprises implique une prise d'état régulière, et un enregistrement de certaines données permettant le rétablissement de l'état global de l'application pour retrouver un état cohérent. De plus, on a vu qu'un tel protocole nécessite un support de l'environnement d'exécution de l'application parallèle.

D'un autre côté, le rétablissement de l'état d'un processus en utilisant des informations contenues dans les processus survivants de l'application est une approche très spécifique à l'application, qui n'est pas transparente puisqu'elle nécessite une participation active de l'application elle-même. Il s'agit d'une approche sollicitant l'application pour qu'elle prenne en charge sa tolérance aux pannes.

Ce chapitre présente deux exemples de mécanismes de tolérance aux pannes pour applications communiquant par passage de messages et leurs performances à grande échelle. Le premier exemple utilise un protocole de retour sur points de reprise coordonnés 4.4, et sollicite relativement peu l'environnement d'exécution. Nous allons voir que l'approche coordonnée, si elle présente des avantages liés à sa simplicité, n'est pas une option qui peut être considérée à grande échelle. C'est pourquoi un mécanisme de retour sur points de reprise non coordonnés est ensuite étudié, afin de ne pas avoir à subir le prix de cette coordination à grande échelle 4.5. Cette approche nécessite un support actif de l'environnement d'exécution, qui doit être lui-même tolérant aux pannes. Les mécanismes employés dans le rétablissement de l'état de l'environnement d'exécution sont détaillés dans la section 4.5.2.

4.4 Retour sur points de reprise coordonnés

Les protocoles de retour sur points de reprise coordonnés étudiés ici sont basés sur l'algorithme de Chandy-Lamport [49]. Cet algorithme a été détaillé dans la section 4.3.1.2. Cette étude porte sur la comparaison de deux implémentations possibles de ce protocole : bloquante et non bloquante.

Les deux protocoles ont été implémentés dans une extension de l'environnement de programmation et d'exécution MPICH, et leurs performances ont été comparées dans plusieurs situations d'exécution. Les résultats de cette comparaison ont été publiés dans [62, 36]. Ma contribution s'est concentrée sur un élément de l'environnement d'exécution avec l'enregistreur de points de reprise et son interaction avec les processus.

4.4.1 Protocoles

4.4.1.1 Non-bloquant : *Vcl*

L'implémentation non-bloquante, appelée *Vcl*, est une implémentation directe de l'algorithme décrit par Chandy et Lamport. Dans l'implémentation de la bibliothèque de communications MPICH, un processus MPI est composé de deux processus : le processus de calcul (le processus MPI lui-même), et un démon de communications. Le démon de communications est utilisé pour conserver les messages en train de transiter sur le réseau, et pour rejouer ces messages en cas de besoin lorsqu'un ou plusieurs processus doivent redémarrer.

L'environnement d'exécution de MPICH est étendu avec l'ajout d'un ordonnanceur de points de reprise, qui est le processus à l'origine des vagues de points de reprise. Des serveurs de points de reprise sont utilisés pour conserver les points de reprise de tous les processus. Enfin, le dispatcher est utilisé pour démarrer, surveiller et redémarrer l'application en cas de défaillance. Le dispatcher a un rôle différent dans le cas d'un environnement tolérant aux pannes selon une approche coordonnée du rôle qu'il a dans un environnement non tolérant aux pannes, puisqu'il doit prendre en charge la réaction à la panne. Concrètement, il doit avoir un comportement différent lorsqu'il détecte une panne : il doit alors non seulement finaliser l'exécution en cours, mais ensuite redémarrer l'application depuis la dernière vague de points de reprise.

Le fonctionnement de cet algorithme est représenté par le schéma de la figure 4.3, en utilisant la représentation de Lamport pour symboliser l'exécution des processus. Le processus MPI de rang 1 reçoit un marqueur venant

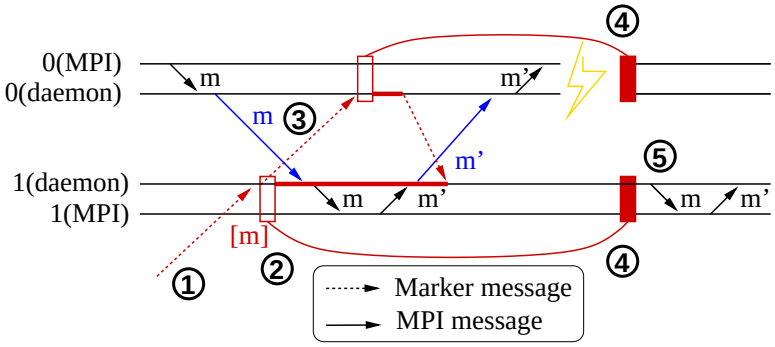


FIGURE 4.3 – *Protocole Vcl : prise d'état en présence de messages, journalisation des messages*

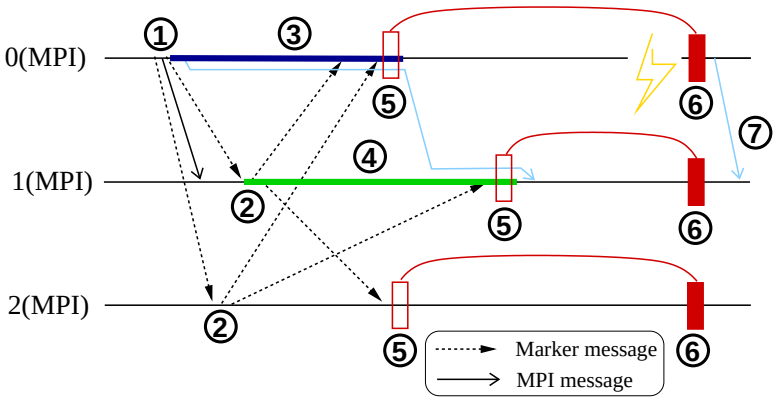


FIGURE 4.4 – *Protocole Pcl : prise d'état en présence de messages, retard d'envoi des messages*

de l'ordonnanceur de points de reprise (1), enregistre son état local dans un instantané (2) et envoie un marqueur à tous les autres processus (2). À partir de ce moment-là, tous les messages reçus par le processus 1 après la prise de son point de reprise local et avant la réception du marqueur provenant de l'expéditeur du message, comme c'est le cas pour le message m, sont enregistrés par le démon de communications. Lorsque le processus MPI de rang 0 reçoit ce marqueur, il enregistre son état local dans un instantané et envoie un marqueur à tous les autres processus (3). La réception de ce marqueur conclut la prise d'état local du processus de rang 1, puisqu'il a reçu les marqueurs de tous les autres processus.

En cas de panne, tous les processus redémarrent à partir du dernier instantané enregistré faisant partie d'une vague qui s'est achevée (4) et les démons de communications rejouent les messages enregistrés (5). Le message m' n'a pas à être rejoué par le protocole de rejeu des messages transitant pendant la vague de points de reprise, car il a été envoyé par le processus de rang 1 après sa prise d'état et le sera (ou pas) renvoyé de toute façon par l'exécution normale du processus.

4.4.1.2 Bloquant : *Pcl*

L'algorithme de l'implémentation non bloquante, appelée *Pcl*, est représenté par le schéma de la figure 4.4. C'est un protocole similaire à celui utilisé par LAM-MPI [148] et OpenMPI [104]. Ce protocole synchronise les processus durant la vague de points de reprise en vidant leurs canaux de communication. Ainsi, aucun message n'est échangé pendant une vague de points de reprise, et l'état des canaux de communications n'a pas à être enregistré. Seule l'image du processus contenue dans l'instantané doit être rechargée pour redémarrer après une défaillance ; les canaux de communications ayant été vidés lors de la vague de points de reprise, ils doivent uniquement être réinitialisés.

Le fonctionnement de cet algorithme est représenté par le schéma de la figure 4.4, en utilisant la représentation de Lamport pour symboliser l'exécution des processus. La circulation du marqueur permet de vider tous les

canaux de communications et de synchroniser les processus. Ce protocole ne fait pas appel à un dispatcher explicite : les vagues de points de reprise sont toujours initiées par le processus de rang 0. Il démarre une vague de points de reprise à intervalles réguliers et passe à l'état *checkpointing*, et il envoie un marqueur à tous les autres processus (1). Quand un processus reçoit un premier marqueur, il passe à l'état *checkpointing*, et il envoie un marqueur à tous les autres processus (2). Après l'envoi d'un marqueur, les processus n'envoient pas d'autre message par ce canal de communication jusqu'à la fin de l'enregistrement de son état (ligne bleue 3). Les messages en attente sont alors toujours dans l'espace mémoire du processus et sont enregistrés dans l'instantané. De même, après la réception d'un marqueur, un processus ne peut pas recevoir de message par le même canal : les réceptions de messages sont retardées jusqu'à la fin de la prise d'état du processus local (ligne verte 4).

Quand un processus a reçu les marqueurs de tous les autres processus, il enregistre son état et envoie le point de reprise au serveur de points de reprise (5). Une fois son état enregistré, les processus sortent de l'état *checkpointing* et peuvent à nouveau envoyer et recevoir des messages. Lorsque son image a été entièrement enregistrée, chaque processus envoie un message au processus de rang 0 pour lui signifier la fin de sa prise d'état, et reprend son exécution. Enfin, le processus de rang 0 envoie un acquittement aux serveurs de points de reprise afin d'assurer la cohérence de la vague de points de reprise une fois qu'il a reçu confirmation de la fin de la prise d'état de chacun des processus.

En cas de panne, tous les processus redémarrent depuis le dernier instantané enregistré (6) et tous les messages dont l'émission a été retardée peuvent être à nouveau envoyés après le redémarrage.

4.4.1.3 Implication de l'environnement d'exécution dans ces protocoles

Ces deux protocoles effectuent une prise de points de reprise *coordonnée*. Ils ont donc besoin tous les deux d'un serveur de points de reprise, qui recevra les points de reprise à chaque vague d'enregistrement de l'état des processus, mais n'effectuent pas de journalisation des messages en-dehors de la vague de points de reprise pour le protocole non bloquant : ils n'ont donc pas besoin de composants tels qu'un enregistreur d'évènements ou une mémoire de canal.

4.4.2 Implémentation

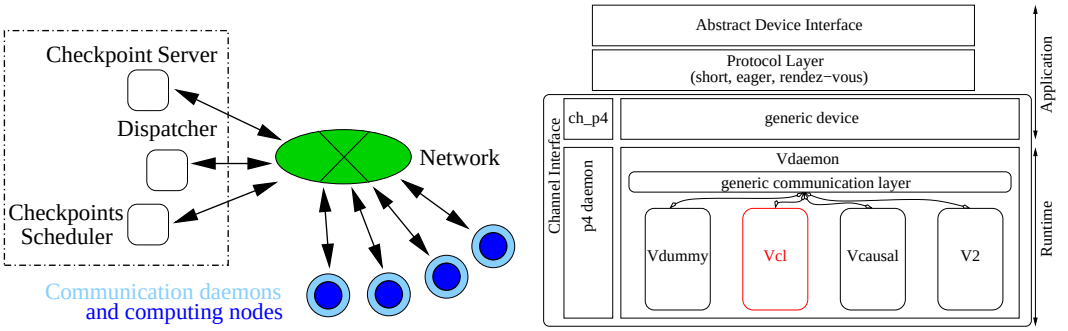
Ces deux protocoles ont été implémentés dans l'implémentation de la norme MPI MPICH [99] grâce au framework développé par le projet MPICH-V [21, 119]. Ce framework a pour but de permettre la comparaison de protocoles de tolérance aux pannes dans les applications MPI, et propose ainsi une architecture générique permettant d'implémenter ces mécanismes de manière équitable. Le protocole *Vcl* a été mis en place dans MPICH-1.2.7 et le protocole *Pcl* a été mis en place dans MPICH2 1.0.3, une version plus récente permettant l'implémentation des fonctionnalités introduites par la norme MPI 2.

4.4.2.1 Architecture

MPICH-*Vcl* L'infrastructure de communications de MPICH repose sur la notion de *canal*[97]. Un canal implémente des routines de communication de bas niveau pour un type de réseaux donné, ou pour un protocole de communications particulier. Le canal de communications pour réseaux TCP est le canal ch_p4. C'est sur lui que se basent les canaux qui implémentent les mécanismes de tolérance aux pannes étudiés dans le framework MPICH-V. Le canal utilisé par MPICH-V est appelé ch_v. L'architecture de la pile de communications de MPICH est représentée sur la figure 4.5(b).

L'infrastructure servant de support au mécanisme de tolérance aux pannes est représenté figure 4.5(a). L'environnement d'exécution de MPICH est étendu avec l'ajout des composants nécessaires à la mise en place du protocole utilisé, c'est-à-dire le serveur de points de reprise, le dispatcher et l'ordonnanceur de points de reprise.

Les communications dans MPICH-V sont effectués en séparant l'application MPI du système de communications, en utilisant des démons de communication (appelés *Vdaemon*) qui effectuent toutes les communications comme exigé par le protocole de tolérance aux pannes utilisé et en utilisant les composants de l'environnement d'exécution nécessaires.



(a) Environnement d'exécution de MPICH-*Vcl*

(b) Intégration du pilote responsable de la tolérance aux pannes dans l'architecture de MPICH.

FIGURE *4.5 – Intégration de Vcl dans MPICH.*

Démons de communications Le démon de communications ne fait pas partie de l'environnement d'exécution à proprement parler, mais il est situé directement à l'interface avec l'environnement d'exécution. Il gère les communications entre les nœuds, c'est-à-dire les envois, réceptions, mises en ordre de messages et établissements de connexions. Une socket TCP est ouverte vers chaque processus MPI et une autre vers chaque démon de l'environnement d'exécution (ici, le dispatcher et le serveur de points de reprise). Il communique avec son processus MPI local au moyen d'une socket Unix.

Dispatcher Le dispatcher remplit les fonctions de base de l'environnement d'exécution : il déploie le reste de l'environnement (serveurs de points de reprise), puis lance l'application sur les nœuds disponibles. Durant l'exécution, il détecte les défaillances en surveillant les fermetures de sockets inattendues, et redémarre l'application. La détection des défaillances est basée sur les paramètres TCP keep-alive du système d'exploitation des machines.

Serveurs de points de reprise et mécanisme de prise de point de reprise Le framework MPICH-V permet d'utiliser différentes bibliothèques d'enregistrement de l'état d'un processus : Condor Standalone Checkpointing Library [121], libckpt [136] et Berkeley Linux Checkpoint/Restart [107, 148]. Ces trois bibliothèques permettent d'enregistrer une image (instantané) d'un processus sur un disque, et de redémarrer ce processus au point où il en était au moment de la dernière prise d'état. L'enregistrement de l'état d'un processus revient à enregistrer l'état de la pile, des registres, de la mémoire virtuelle du processus et la valeurs du compteur d'instructions. Le rechargement en mémoire de ces éléments permet de repartir dans l'état sauvegardé du processus. La bibliothèque utilisée dans cette étude est BLCR.

Les points de reprise de tous les processus sont transférés au serveur de points de reprise pour être enregistrés de manière fiable, et peuvent être transférés du serveur de points de reprise afin de redémarrer un processus de l'application sur une nouvelle machine après une défaillance.

Pour effectuer une prise de point de reprise, un processus MPI se duplique en utilisant l'appel système `fork()`. Le processus fils effectue la prise de point de reprise et l'écrit dans un fichier, pendant que le père continue son exécution. Le démon de communications du processus MPI établit une connexion avec le serveur de points de reprise, et envoie l'image du processus au serveur au fur et à mesure que l'image du processus est écrite dans le fichier.

Pendant la prise et le transfert de l'image du processus, les messages échangés sont sauvegardés en accord avec le protocole, dans la mémoire volatile du démon afin de pouvoir les envoyer au serveur de points de reprise de la même manière. Les images les plus récentes des processus locaux et des messages sont conservés sur le disque local afin de pouvoir redémarrer à partir de l'image locale en cas de défaillance d'un autre processus, et de ne pas avoir à demander au serveur de points de reprise de renvoyer les images à tous les processus.

Ce mécanisme permet de ne pas avoir à interrompre l'exécution du processus MPI lui-même durant la phase de prise d'état.

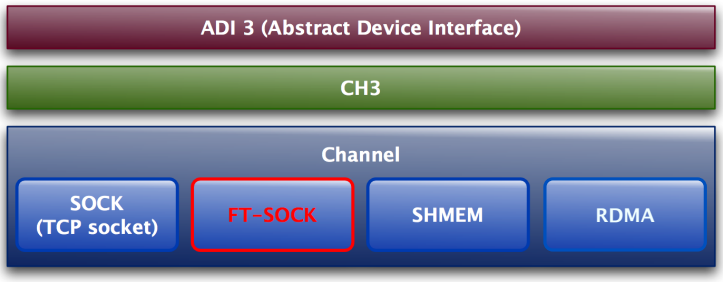


FIGURE *4.6 – Intégration du pilote responsable de la tolérance aux pannes dans l'architecture de MPICH2.*

Ordonnanceur de points de reprise Les vagues de points de reprise sont orchestrées par l'ordonnanceur de points de reprise, qui envoie des marqueurs aux processus MPI à intervalles réguliers (définis par l'utilisateur) et signifie aux serveurs de points de reprise la fin des vagues de points de reprise une fois qu'il a reçu les marqueurs de tous les processus MPI.

MPICH-*Pcl* Le protocole *Pcl* a été implémenté dans MPICH2, une nouvelle implémentation de la norme MPI succédant à MPICH afin de permettre l'intégration des nouvelles fonctionnalités, notamment dynamiques, introduites par la norme MPI 2. MPICH2 est composé de trois couches :

- l'interface abstraite de canal (ADI3, ou *Abstract Device Interface*), qui implémente les routines de communication MPI en utilisant un ensemble de routines de communications de haut niveau ;
- le canal de communications CH3, qui présente à ADI3 une interface composée de quelques routines de communications (entre dix et vingt) ;
- un canal de communications qui implémente les fonctions de CH3 spécifiquement pour un type de réseaux ou un protocole de communications.

Le canal de communications pour réseaux TCP est appelé `sock` ; il a servi de base au canal utilisé pour l'implémentation de la tolérance aux pannes, appelé `ft-sock`. Un autre canal a été utilisé pour permettre l'utilisation des réseaux à hautes performances : le canal `Nemesis` [37], utilisant un segment de mémoire partagée pour faire communiquer des processus situés sur la même machine et un réseau à hautes performances pour communiquer entre machines différentes.

L'environnement d'exécution de MPICH2 a été étendu en introduisant un serveur de points de reprise similaire à celui utilisé par le protocole *Vcl*.

Contrairement au protocole *Vcl*, les vagues de prises d'état ne sont pas démarrées par un ordonnanceur de points de reprise. Elles sont toujours démarrées par le processus MPI de rang 0.

La prise d'état se fait ici aussi en clonant le processus. Les mécanismes de communications de bas niveau doivent être éteints, c'est-à-dire que les sockets réseau doivent être fermées et les segments de mémoire partagée liés aux communications (mémoire des cartes réseaux qui en utilisent, comme Myrinet ou InfiniBand, et mémoire partagée utilisée pour communiquer entre processus) doivent être démappés.

Prise d'état et enregistrement du point de reprise Le mécanisme d'enregistrement de l'état des processus est similaire à celui utilisé pour MPICH-*Vcl*.

Environnement d'exécution : MPD et FTPM L'environnement d'exécution global de MPICH2 est appelé MPD [39]. Il est exécuté sur tous les nœuds du système au moyen d'un processus exécuté de manière persistante et qui est chargé de lancer les applications MPI. Les démons sont interconnectés selon un anneau. MPD est plutôt un lanceur, évitant ainsi d'avoir à lancer une campagne de `ssh` pour lancer une application.

Le lancement d'une application est fait en clonant des processus du MPD pour lancer les PM, ou *Process Managers*. L'ensemble des PM forme l'environnement d'exécution rattaché à une application MPI donnée, tandis que MPD est utilisé par toutes les applications MPI exécutées sur le système. MPD a un rôle de deployeur uniquement, et les PM prennent en charge le reste des tâches de l'environnement d'exécution. Ils lancent

notamment les processus de l'application MPI en se clonant, et les mettent en relation les uns avec les autres lorsqu'une communication doit être effectuée. Dans l'implémentation de base de MPICH, MPD est tolérant aux pannes. Cependant, les PM ne le sont pas et en cas de défaillance dans une application, tous les PM et les processus MPI sont tués.

MPD lance uniquement les PM, mais ne peut pas lancer d'éléments supplémentaires de l'environnement d'exécution comme les serveurs de points de reprise. Le lanceur de MPICH2 a donc été entièrement réécrit pour *Pcl* afin de pouvoir intégrer le support de la tolérance aux pannes. Ce nouvel environnement d'exécution s'appelle FTPM (*Fault Tolerant Process Manager*). FTPM ne contient pas de démons MPD. Il remplit les rôles essentiels d'un environnement d'exécution en implémentant l'interface PMI décrite dans la section 2.1.1.2 du chapitre 2 directement dans le `mpiexec`. FTPM prend en charge le mécanisme de tolérance aux pannes en détectant les défaillances et en redémarrant l'application à partir des points de reprise sauvegardés. Il est constitué d'un `mpiexec` qui sert de point de lancement et de démons modifiés à partir des PM. Le format du fichier machine donné en entrée du `mpiexec` a été étendu afin de prendre en charge les serveurs de points de reprise et de les attribuer à des machines en particulier. La détection des défaillances est effectuée de façon similaire à MPICH-*Vcl*.

La mise en relation des processus les uns avec les autres se fait en publiant une carte de visite associée à chaque rang de processus. La carte de visite est composée de l'adresse IP, du nom DNS de la machine et du port sur lequel le processus peut être contacté. Une base de données distribuée est maintenue par le FTPM, contenant les cartes de visites de tous les processus ainsi que la localisation des points de reprise sur les différents serveurs de points de reprise et le numéro de la dernière vague de points de reprise effectuée complètement, afin de permettre au processus victime de la panne de pouvoir localiser le point de reprise qu'il doit obtenir pour redémarrer sur une nouvelle machine. Les autres processus étant relancés sur la même machine, ils peuvent repartir d'un point de reprise disponible sur leur machine locale.

4.4.3 Performances

Les performances de ces protocoles ont été évaluées et comparées dans trois environnements matériels typiques :

- une grappe interconnectée par réseau Ethernet Gigabit ;
- une grappe interconnectée par réseau rapide Myrinet ;
- une grille de calcul à grande échelle.

Les grappes utilisées font partie de la plate-forme expérimentale Grid'5000 [41]. Nous avons utilisé six grappes, toutes équipées de deux micro-processeurs AMD Opteron 248 cadencés à 2 GHz. Les grappes utilisées sont les suivantes :

- Bordeaux : 48 nœuds
- Lille : 53 nœuds
- Orsay : 216 nœuds (utilisée pour les expériences sur grappe avec réseau Ethernet)
- Rennes : 64 nœuds
- Sophia-Antipolis : 105 nœuds (utilisée pour les expériences sur grappe avec réseau rapide)
- Toulouse : 58 nœuds

Chaque grappe dispose d'un réseau Ethernet Gigabit, et la grappe de Bordeaux dispose également d'un réseau Myrinet2000. Tous les nœuds tournent sous un noyau Linux 2.6.13.5, et tous les exécutables sont compilés par GCC-4.0.3 avec l'optimisation O3.

Les bancs de tests utilisés sont issus de la suite NAS NPB2.3 [16]. Les prises de points de reprise sont déclenchées à intervalles de temps réguliers. Pour ces expériences, nous avons utilisé des intervalles de l'ordre de quelques dizaines de secondes afin d'effectuer plusieurs vagues de prises de points de reprise pendant une exécution. Cependant, en conditions réelles, les prises de points de reprise sont plus espacées.

4.4.3.1 Grappe a réseau Ethernet Gigabit

La figure 4.7 présente une étude de scalabilité en fonction du nombre de serveurs de points de reprise. Nous avons exécuté le banc de test BT de classe B sur 64 processeurs (32 nœuds bi-processeurs), avec un intervalle entre deux vagues de points de reprise fixé à 30 secondes. Une fois l'instantané complètement transféré vers le serveur de points de reprise, l'ordonnanceur de points de reprise attend 30 secondes avant de démarrer une

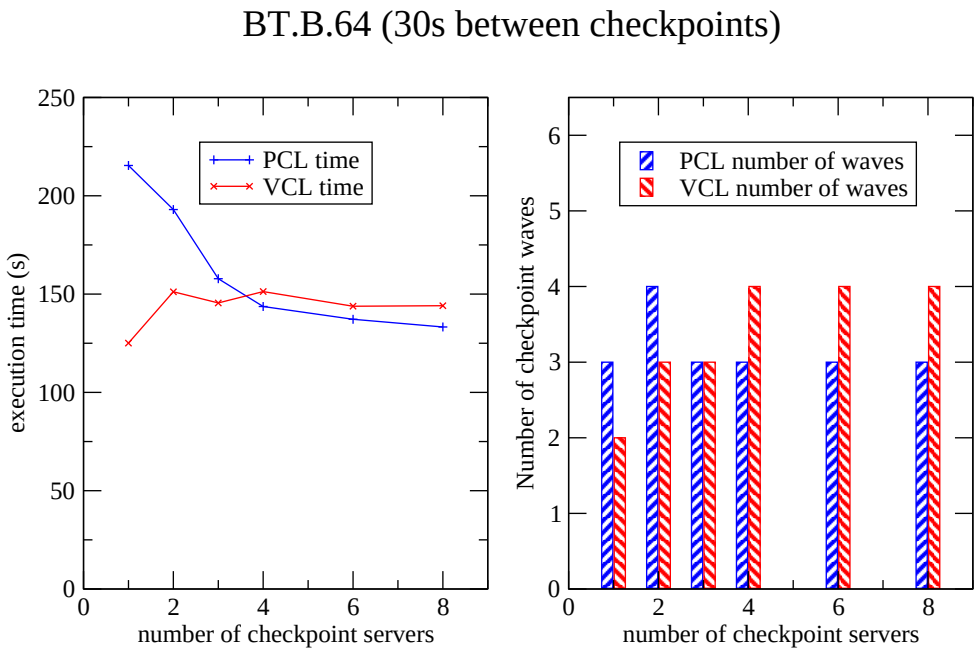


FIGURE *4.7 – Impact du nombre de serveurs de points de reprise sur un banc de test BT de classe B pour 64 processus en utilisant un intervalle de temps donné entre les vagues de points de reprise.*

nouvelle vague de points de reprise. La partie gauche de la figure montre le temps d'exécution pour différents nombres de processus par serveur de points de reprise, d'un serveur pour 64 nœuds à 1 serveur pour 8 nœuds. La partie droite de la figure montre le nombre de vagues de points de reprise effectuées pour ces exécutions.

On constate que le temps d'exécution de *Vcl* reste à peu près constant quel que soit le nombre de serveurs de points de reprise utilisé, tandis que le temps d'exécution de *Pcl* décroît quand le nombre de serveurs de points de reprise augmente. La diminution du nombre de processus par serveur de points de reprise permet, en diminuant la charge sur les serveurs de points de reprise, de diminuer le temps de l'enregistrement des points de reprise. Le protocole *Pcl* requiert de bloquer les communications pendant la prise du point de reprise, et de les reprendre pendant l'enregistrement du point de reprise sur le serveur. Les communications de l'application sont donc en compétition avec le transfert du point de reprise. Si la contention au niveau du serveur de points de reprise est diminuée, ce temps de transfert est réduit et la perturbation des communications est diminuée. Cette amélioration est bien visible au début de la courbe, pour un petit nombre de serveurs de points de reprise et donc une contention forte à ce niveau. Le temps d'attente entre deux vagues de point de reprise est redémarré dès que tous les processus ont transféré leur instantané. C'est pourquoi l'augmentation du nombre de serveurs de points de reprise diminue le temps d'attente entre deux vagues de points de reprise, cependant le temps total d'exécution diminue suffisamment pour ne pas déclencher une vague de points de reprise supplémentaire.

À l'inverse, *Vcl* ne bloque pas les communications pendant la vague de points de reprise, donc moins de communications ont lieu après la prise du point de reprise et pendant le transfert vers les serveurs de points de reprise : les communications sont donc moins perturbées par le transfert du point de reprise. Le déclenchement d'une vague de points de reprise supplémentaire, provoquée par le temps gagné entre deux vagues, n'a alors pas d'influence significative sur le temps d'exécution. Le protocole *Vcl* perturbe moins les communications MPI que

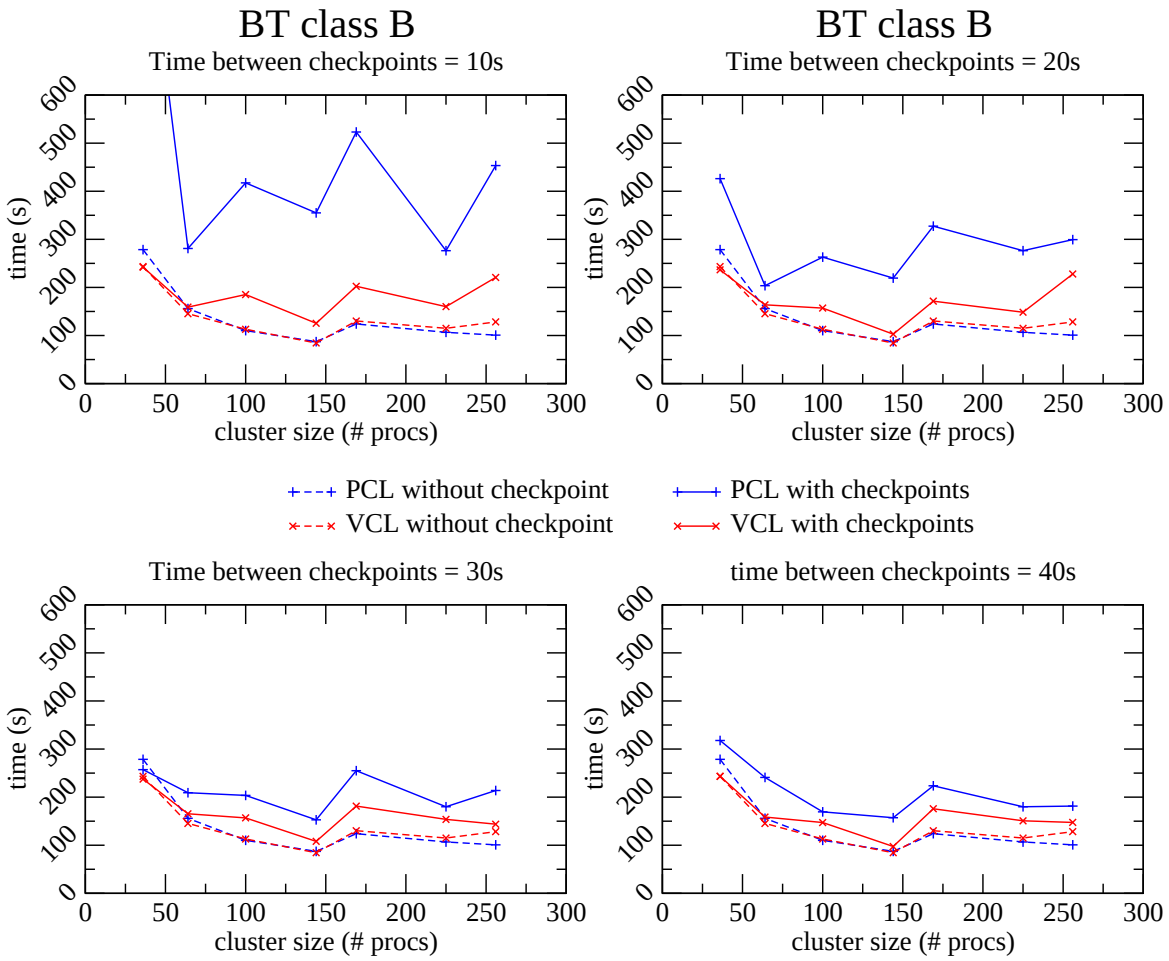


FIGURE *4.8 – Impact de la fréquence d'enregistrement des points de reprise en fonction du nombre de processus.*

Pcl, et on voit que l'augmentation du nombre de vagues de points de reprise n'altère pas le temps d'exécution. La différence de performances entre les deux courbes pour six et huit serveurs de points de reprise s'explique par la légère différence de performances entre MPICH et MPICH2.

On constate donc que la charge sur l'environnement d'exécution et en particulier le serveur de points de reprise se manifeste de manière plus importante avec l'implémentation bloquante, où les processus sont synchronisés pendant l'enregistrement du point de reprise. L'implémentation non bloquante permet plus de lattitude dans la prise des points de reprise, et est moins sensible à la performance de l'environnement d'exécution.

La figure 4.8 montre la scalabilité des protocoles de tolérance aux pannes pour différents temps d'attente entre deux vagues de points de reprise. Nous avons exécuté le banc de test BT de classe B sur un nombre variable de processeurs, en variant également le délai entre deux vagues de points de reprise, en utilisant toujours le même nombre de serveurs de points de reprise (9).

En l'absence de prises de points de reprise, les deux implémentations montrent des performances similaires, MPICH2 étant légèrement plus performant sur 256 processeurs. Le ralentissement observé pour toutes les mesures à 169 processus s'explique par le fait que n'ayant que 150 machines à notre disposition pour ces mesures, nous avons utilisé deux processus par machine (bi-processeur) pour des tailles supérieures à 160 processus et un seul jusqu'à 144 processus. La carte réseau est alors partagée par les deux processus.

La courbe pour une vague de points de reprise toutes les 10 secondes montre un système fortement perturbé par les prises de points de reprise du protocole bloquant. Sur les autres courbes, on peut voir que pour les deux protocoles, l'augmentation du nombre de processus n'a pas d'influence significative sur le surcoût induit par la tolérance aux pannes. L'augmentation de la fréquence des prises de points de reprise sature l'environnement

d'exécution qui ne peut plus fournir une qualité de service acceptable pour l'application et dégrade fortement les performances du protocole bloquant, qui passe la plupart de son temps à effectuer des synchronisation lorsque la fréquence est élevée, et les performances des communications sont souvent dégradées. Le protocole *Vcl* n'implique pas de synchronisation, et subit moins les conséquences de la saturation de l'environnement d'exécution en permettant à l'application de pouvoir continuer à avancer pendant la vague de prise de points de reprise. Un espacement plus important des vagues de points de reprise limite la différence entre les deux protocoles.

4.4.4 Discussion

Cette étude sur le retour sur points de reprise coordonnés montre qu'un protocole relativement simple et ne demandant pas une grande implication de l'environnement d'exécution (pour l'enregistrement des points de reprise uniquement) implique une synchronisation pénalisante à grande échelle, en rendant les phases de prises de points de reprise très longues. Ces protocoles peuvent être améliorés en étant moins impératifs : alors qu'une synchronisation ferme impose à tous les processus d'arrêter leur exécution pour effectuer leur prise de point de reprise local et l'enregistrer auprès de l'environnement d'exécution, il est possible d'assouplir cette synchronisation en permettant aux processus de poursuivre leur exécution durant la vague de prises de points de reprise, qui ne sont alors plus effectuées strictement de façon simultanée, réduisant ainsi la contrainte imposée à l'environnement d'exécution. Cependant ces protocoles non bloquants induisent quand même une synchronisation, même si elle est plus souple que dans le cas des protocoles bloquants.

Des études statistiques sur la fiabilité des systèmes à grande échelle en fonction du nombre de composants utilisés [145] montrent que le temps moyen avant défaillance diminue rapidement quand le nombre de composants augmente. Pour des systèmes composés de l'ordre de quelques milliers de nœuds il est de quelques heures.

Si des optimisations permettent de réduire le temps de lancement, comme vu dans le chapitre 3, et un dimensionnement adéquat de l'environnement d'exécution permet de réduire la durée d'une vague de points de reprise (jusqu'à un temps considéré comme "optimal"), à grande échelle il demeure obligatoirement un temps incompressible de (re)démarrage de l'application et de prise de la vague de points de reprise suivante. Si ce temps est inférieur au temps moyen avant défaillance du système, l'application ne peut plus avancer jusqu'à la vague suivante, et doit toujours redémarrer sur le même ensemble de points de reprise : l'application ne peut plus avancer.

Une expérience sur Grid'5000 utilisant 1 024 nœuds a permis d'estimer le temps d'effectuer une vague de points de reprise à cette échelle à environ trente minutes. De même, le déploiement de l'environnement d'exécution et le lancement d'une application avec MPICH2 et l'environnement d'exécution modifié pour supporter la tolérance aux pannes dure environ trente minutes.

Dans l'exemple où le démarrage d'une application et une vague de points de reprise prennent trente minutes chacun, le temps moyen avant défaillance doit être supérieur à une heure.

On est alors face à un mur empêchant la terminaison de l'application : le système est trop grand pour être suffisamment fiable par rapport au temps de lancement et d'enregistrement de l'état de l'application, ou trop grand pour lancer et enregistrer l'application assez rapidement au regard de la fiabilité du système. Cette situation est illustrée par la figure 4.9.

À grande échelle, l'approche coordonnée n'est alors plus raisonnable. Il faut se tourner vers des stratégies de retour sur points de reprise non coordonnés, et un support plus important de l'environnement d'exécution. Les protocoles de retour sur points de reprise imposent seulement aux processus victimes de la panne de repartir en arrière, et laissent les autres processus continuer leur exécution. Le maintien de la cohérence de l'état impose de sauvegarder les messages échangés, ce qui complexifie le chemin critique des communications en imposant des copies supplémentaires et le rend moins efficace à petite échelle qu'une stratégie coordonnée, qui n'introduit pas de surcoût en-dehors des prises de points de reprise, comme on l'a vu ci-avant.

De plus, les protocoles de prise de points de reprise coordonnés imposent le moment de la prise du point de reprise à chaque processus, et d'autant plus lorsqu'ils sont bloquants. Le serveur de points de reprise est alors sollicité par tous les processus en même temps, ce qui sature d'autant plus l'environnement d'exécution. On a vu que dans l'implémentation bloquante, l'application ne pouvant pas reprendre ses communications tant que l'enregistrement de son point de reprise n'était pas terminé, la saturation de l'environnement d'exécution avait un impact important sur les performances de l'application. À l'inverse, les protocoles non coordonnés introduisent

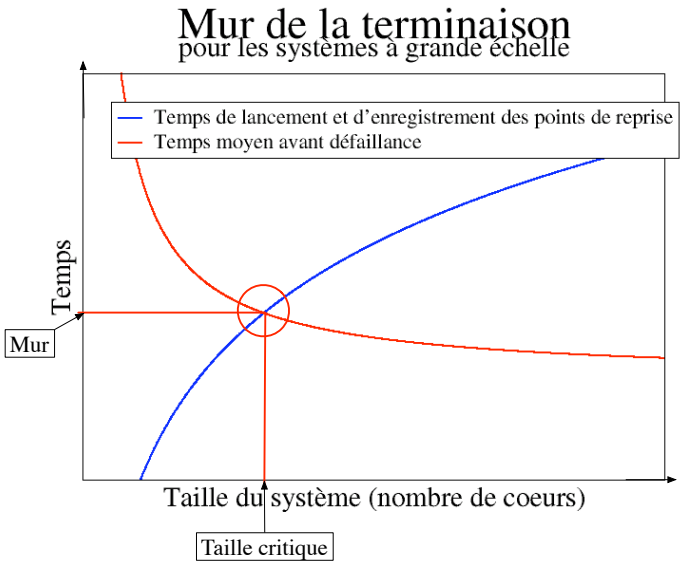


FIGURE *4.9 – Mur de la terminaison d'une application : à partir d'une certaine taille de système, le temps de lancement de l'application et de prise de la prochaine vague de points de reprise excède le temps entre deux pannes, et l'application ne peut plus avancer.*

un ordonnanceur de points de reprise qui permet de mettre en place des stratégies d'ordonnancement de ces prises de points de reprise, afin (entre autres) de les étaler plus dans le temps.

Les stratégies non-coordonnées nécessitent une participation active de l'environnement d'exécution dans la tolérance aux pannes. En particulier, celui-ci ne doit pas seulement être conscient de la tolérance aux pannes : il doit lui-même être tolérant aux pannes, et être capable de restaurer son état pour supporter le protocole qui restaure l'état de l'application.

4.5 Support des protocoles de retour sur points de reprise non coordonnés

Les protocoles de tolérance aux pannes basés sur des retours sur points de reprise non coordonnées, comme par exemple les protocoles de journalisation de messages, ont besoin d'une participation plus importante de l'environnement d'exécution. On a vu dans la section 4.4 qu'un protocole à points de reprises coordonnés peut ne demander de l'environnement qu'une extension pour sauvegarder de façon stable les points de reprise, et une conscience de la tolérance aux pannes dans le sens où il doit être capable de redémarrer l'application après une panne à partir des points de reprise et non pas à partir du point initial de l'application.

4.5.1 Intégration dans le framework OpenMPI-V

Ce travail a vocation à s'intégrer principalement dans le cadre du framework d'OpenMPI pour la tolérance aux pannes non-coordonnée [27], comme l'implémentation d'un protocole de journalisation de messages optimisée [28]. J'ai donc conçu un protocole permettant à un environnement d'exécution de s'auto-cicatriser en tenant compte des spécificités du rôle qu'il remplit auprès de l'application. J'ai ensuite implémenté ce protocole dans ORTE, l'environnement d'exécution d'OpenMPI.

Indépendamment, la tolérance aux pannes au niveau de l'application est mise en place dans un framework au niveau de la bibliothèque de communications et éventuellement de composants ajoutés à l'environnement d'exécution sous forme d'*outils*. Le framework au niveau de la bibliothèque de communications fonctionne en s'interposant dans l'architecture modulaire d'OpenMPI afin de ne pas perturber le fonctionnement interne des

autres composants tout en agissant *entre* les composants de la pile de communications. Cette interposition est représentée par la figure 4.10, et agit au niveau des routines de communications de haut niveau (dans le PML, le bas niveau, spécifique à chaque type de réseau, étant implémenté dans les BML). Ce composant vampire, appelé *PML-V*, n'implémente aucune des fonctions de communications elles-mêmes, mais appelle ces fonctions dans un composant PML dit *hôte* qui effectuera effectivement ces communications. Le PML-V est donc un framework générique pour implémenter des actions à effectuer sur les messages envoyés, comme de la journalisation ou des modifications (ajout en queue, par exemple), à haut niveau dans la pile de communications, c'est-à-dire avant que les messages ne soient éventuellement découpés pour être envoyés sur les interfaces réseau disponibles.

L'extension de l'environnement d'exécution est faite de la même façon que pour les protocoles de points de reprise non-coordonnés avec journalisation des messages usuels [21, 29, 31, 32, 119, 30] décrits dans la section 4.3.1.3 : les points de reprise sont enregistrés sur un serveur de points de reprise supposé fiable et leur prise est déclenchée par un ordonnanceur de points de reprise, les déterminants permettant de reconstituer l'ordre causal des messages sont sauvegardés sur un enregistreur d'évènements supposé fiable.

FIGURE *4.10 – Architecture d'OpenMPI-V : placement du PML Vampire dans la pile de communications OMPI*

4.5.1.1 Communications au sein de l'environnement d'exécution

Les communications nécessaires au protocole de tolérance aux pannes sont des messages hors-bande, par définition de ceux-ci : en effet, ils ne sont pas utiles directement à l'application dans le sens où ils ne sont pas dictés explicitement par des communications MPI, mais ils sont nécessaires à son exécution. Dans OpenMPI, les communications hors-bande sont implémentées dans l'environnement d'exécution et ne supportent que les réseaux TCP. En effet, les performances de ces communications sont moins critiques que celles des communications de l'application, et le choix d'utiliser TCP permet de ne pas perturber les communications de l'application si celles-ci utilisent des réseaux rapides.

Lorsqu'un évènement non-déterministe doit être enregistré par l'environnement d'exécution, cet enregistrement engendre une communication entre le processus MPI et l'enregistreur d'évènements (en réalité, un aller-retour). Bien que l'enregistrement soit effectué de manière asynchrone pendant la délivrance du message au processus, comme décrit dans [28], il doit être terminé au moment où le message est délivré à l'application. Si les communications entre les processus et l'enregistreur d'évènements sont plus lentes que les communications de l'application, elles risquent de ralentir celles-ci de façon trop importante en limitant leurs performances à celles du réseau TCP. C'est pourquoi ces communications sont exceptionnellement implémentées en utilisant les communications de bas niveau de la bibliothèque MPI.

4.5.1.2 Retour des erreurs au niveau MPI

Les erreurs ne doivent pas être remontées à l'application MPI. En effet, l'application ne doit pas savoir qu'une erreur a eu lieu : l'intergiciel doit rendre la tolérance aux pannes transparente. La conduite à tenir en cas de remontée d'erreur provenant du PML dépend de l'opération qui a échoué. Les fonctions implémentées par le PML V retournent donc toujours une valeur signifiant que l'opération a réussi.

4.5.2 Environnement d'exécution tolérant aux pannes

L'implémentation d'un protocole de retour sur points de reprise non coordonnés nécessite une participation active de l'environnement d'exécution. En effet, il est nécessaire à la fois de s'assurer que les processus survivants continuent leur exécution, et de relancer le processus victime de la panne. On a alors besoin d'un environnement d'exécution résilient capable de se cicatriser de lui-même.

Celui-ci doit être tolérant aux pannes, dans le sens où il doit restaurer son état pour atteindre un état compatible avec celui dans lequel il aurait été si aucune panne n'était survenue. Il n'est pas possible d'envisager une approche de retour sur points de reprise : ces approches nécessitent un support de l'environnement d'exécution, or, on cherche à tolérer les défaillances dans l'environnement d'exécution lui-même. De plus, les contraintes liées à l'enregistrement de ces points de reprise et au retour en arrière sur points de reprise liées au maintien de la cohérence de l'état présentent des contraintes qui ne sont pas compatibles avec le rôle de l'environnement d'exécution. Par exemple, l'arrêt de l'exécution d'un processus de l'environnement d'exécution pour la prise d'un point de reprise équivaut à une interruption temporaire du service fourni à l'application et pourrait avoir des conséquences sur les performances du processus MPI ; un retour en arrière coordonné de tous les processus de l'environnement d'exécution aurait des conséquences sur l'application, qui devrait vraisemblablement retourner en arrière elle aussi ; enfin, l'environnement d'exécution doit continuer à remplir son rôle durant la phase de restauration de son état, au moins auprès des processus toujours en vie.

4.5.2.1 Définition de l'état d'un environnement d'exécution

L'état d'un environnement d'exécution est rétabli dès lors qu'il est capable d'assurer à nouveau ses fonctions en mode non-dégradé. Comme vu dans la section 1.2 du chapitre 1, il doit alors être en mesure de :
- communiquer, au sein de l'environnement d'exécution ou avec les processus de l'application ;
- lancer des processus, surveiller l'état des processus lancés, terminer les processus à la fin de l'exécution ;
- permettre aux processus de l'application de communiquer entre eux.

Infrastructure de communications L'état de l'infrastructure de communications fait partie de l'état de l'environnement d'exécution. La connexité du graphe définissant la topologie de communications doit être rétablie, dans un état défini par les spécifications de cette topologie : le graphe peut être différent de celui qui était en place avant la panne, ou il peut devoir être exactement identique, par exemple pour préserver des propriétés particulières.

Application L'application doit être dans un état lui permettant de retrouver son état d'avant la panne, en utilisant un mécanisme de tolérance aux pannes. Le rétablissement de l'état de l'application est distinct du rétablissement de l'état de l'environnement d'exécution. En particulier, le nombre de processus de l'application doit être identique au nombre de processus exécutés avant la panne [3].

Communications de l'application L'environnement doit permettre de rétablir les communications de l'application, et en particulier, entre les processus victimes de la panne et les processus survivants.

Nombre de processus Le nombre de processus composant l'environnement d'exécution peut être considéré comme un paramètre représentatif de l'état de l'environnement d'exécution, mais il n'a pas forcément à être rétabli. Les processus de l'environnement d'exécution gèrent chacun un nombre variable de processus de l'application. Par exemple, on peut décider d'exécuter n processus d'une application sur n machines. On aura alors un environnement d'exécution composé de n processus. Si une machine tombe en panne, et que l'on ne dispose pas de machine supplémentaire, mais que parmi les machines restantes il en existe au moins une pouvant exécuter deux processus de l'application, on pourra redémarrer le processus victime de la panne sur une machine sur laquelle tourne déjà un processus de l'environnement d'exécution : celui-ci verra alors son état restauré sans que l'on ait ajouté de processus pour remplacer celui qui est tombé en panne, et son état est alors restauré avec un nombre de processus inférieur à celui existant avant la panne.

Le nombre de processus constituant l'application est à l'inverse important, mais le nombre de processus après la restauration de l'état de l'application n'est pas forcément égal au nombre de processus présents avant la panne. Par exemple, si un processus enregistre son état alors que l'application est composée de N processus, puis elle crée dynamiquement M nouveaux processus avant qu'un processus ne soit arrêté par une panne. Le protocole de tolérance aux pannes utilisé fait revenir en arrière tous les processus de l'application : celle-ci se retrouve composée de N processus, qui recréeront plus tard les M processus créés dynamiquement. Dans un

3. À distinguer du nombre de processus au lancement de l'application, de nouveaux processus ayant pu être créés dynamiquement au cours de l'exécution avant la panne.

protocole où seuls les processus victimes de la panne effectuent un retour en arrière, seuls les processus qui ont enregistré leur état sont redémarrés : les autre seront recréés dynamiquement au cours de l'exécution. La dynamicité du nombre de processus pose alors un problème pour les protocoles non-coordonnés, notamment ceux basés sur la journalisation et le rejeu de messages : si un processus est détruit dynamiquement au cours de l'exécution, il ne pourra plus rejouer les messages envoyés à un autre processus.

État de l'environnement d'exécution On peut donc définir l'état d'un environnement d'exécution comme le nombre de processus de l'application qu'il gère, les propriétés de la topologie de son infrastructure de communications, et la connaissance qu'il a des moyens de communications entre les processus de l'application.

4.5.2.2 Qualité de service pendant la phase de rétablissement de l'état

L'impact de la panne sur le reste de l'environnement d'exécution doit être le plus limité possible. Une panne peut avoir un impact sur le reste de l'application (par exemple, blocage des envois de messages), mais le comportement de l'application est déterminé par le protocole de tolérance aux pannes au niveau de l'application.

Les processus de l'environnement d'exécution n'étant pas en contact avec la panne n'en sont pas forcément informés durant le protocole de rétablissement d'état, et continuent leur exécution normalement. Les processus en contact avec la panne sont ceux communiquant directement avec le processus victime de la panne. Dans ce cas, les messages ne peuvent plus être envoyés vers le processus victime de la panne : ils sont mis en attente, et seront envoyés une fois que ce processus sera à nouveau contactable.

Les communications collectives constituent un cas particulier dans les fonctionnalités d'un environnement d'exécution. En effet, elles sont non-bloquantes et basées sur des routines de communications pair-à-pair non-bloquantes. Cependant, elles peuvent constituer un point de synchronisation, par exemple avec une barrière (équivalente à la routine MPI_Barrier). Dans ce cas, les processus survivants attendent que tous les processus soient en état de communiquer pour pouvoir terminer la barrière.

4.5.2.3 Protocole de rétablissement de l'état de l'environnement d'exécution

Le rétablissement de l'état de l'environnement d'exécution se fait en quatre phases après la détection de la panne :

- si nécessaire : lancement d'un nouveau processus, venant prendre la place du processus qui vient de mourir dans l'infrastructure de communications ;
- si on vient de lancer un nouveau processus : rétablissement des communications entre les processus survivants et le processus qui vient d'être relancé, par un échange d'informations de contact ;
- relancement des processus de l'application qui ont été victimes de la panne ;
- rétablissement des communications de l'application en permettant l'échange des informations de contact entre les processus de l'application survivants et les processus venant d'être redémarrés.

On suppose ici l'existence d'un mécanisme de détection de défaillances. Le cas où les processus de l'application sont relancés sur des machines déjà utilisées, et où l'on ne relance pas de nouveau processus de l'environnement d'exécution pour remplacer celui tombé en panne étant un cas particulier, le protocole entier (avec rétablissement du nombre de processus de l'environnement) est décrit dans ce qui suit. De plus, certaines topologies de communications comme les arbres peuvent avoir besoin, soit de remplacer impérativement le processus manquant, soit de mettre en place un protocole permettant de rétablir la topologie en cas de réduction du nombre de processus.

Dans ce qui suit, on désigne comme *démon* un processus de l'environnement d'exécution, et par *processus MPI* un processus de l'application MPI exécutée par l'environnement d'exécution.

Topologie centralisée Dans le cas d'une topologie centrée sur un processus remplissant des fonctionnalités d'orchestration de l'environnement d'exécution (gestion des ressources, point de départ du lancement de l'application, point de convergence des sorties standards et d'entrée de l'entrée standard et des signaux), on utilise un gestionnaire d'erreurs centralisé au niveau du gestionnaire des ressources.

Une fois la panne détectée et rapportée au gestionnaire d'erreurs. Celui-ci obtient une machine supplémentaire disponible pour relancer le démon victime de la panne. Il effectue la réallocation des processus MPI sur ce démon ou sur d'autres, pour restaurer la carte de l'application sur l'environnement d'exécution.

Le lancement du nouveau démon se fait de la même manière que lors du premier lancement de l'application : le démon est lancé, et il rapporte son information de contact au démon central. Celui-ci met à jour les coordonnées du nouveau démon dans l'infrastructure de communications, afin de rétablir les communications au sein de l'environnement d'exécution.

Une fois l'infrastructure de communications rétablie, les processus de l'application victimes de la panne peuvent être relancés. Ils sont relancés de la même façon que lors du lancement initial de l'application, à ceci près que le démon peut choisir de relancer les processus à partir de points de reprise, s'il en existe.

Topologie en arbre Une partie du protocole de rétablissement de l'infrastructure de communications peut être distribuée et non gérée par un même processus central pour tous les processus dans le cas d'une topologie en arbre. Dans ce cas, chaque processus a connaissance et est en contact avec les processus qui lui sont directement connectés dans le niveau supérieur de l'arbre (son processus père) et dans le niveau inférieur (ses processus fils).

La détection de la panne et la correction de celle-ci peut alors être prise en charge par un des voisins du démon victime de la panne. Pour des raisons de connexité, le processus père est en charge de la détection de la panne et du protocole de relancement du nouveau démon.

Il doit d'abord obtenir une nouvelle machine auprès du gestionnaire de ressources. Ce service doit présenter un comportement *cohérent* dans le système, c'est-à-dire qu'il doit savoir, à tout moment, quelles machines sont utilisées et quelles machines sont disponibles et non utilisées. Le fait de distribuer ce service pose des problèmes de cohérence, c'est pourquoi le choix a été fait ici de le centraliser. C'est en cela qu'une partie du protocole de rétablissement de l'infrastructure est distribuée, et non pas son intégralité. De plus, cette centralisation permet une compatibilité avec une éventuelle intégration avec les systèmes de réservations de ressources, dans le cas où il serait possible pour l'environnement d'exécution d'interagir avec le système de réservation pour obtenir des machines supplémentaires.

Le démon chargé de relancer le démon victime de la panne le relance alors comme dans le cas de la topologie centralisée, en jouant localement (par rapport au démon victime de la panne) un rôle similaire à celui du démon central.

Dans une topologie en arbre, les processus de l'environnement d'exécution n'ont pas besoin de disposer de l'intégralité des informations sur le système, mais seulement de celles des processus situés en-dessous d'eux dans l'arbre (leurs *descendants*) et immédiatement au-dessus d'eux (leur *parent*). Ces informations sont composées du nom des machines sur lesquelles sont exécutées les processus, nécessaires au lancement des processus, et leurs informations de communications. Les communications effectuées au sein de l'environnement d'exécution étant routées le long de cet arbre, il n'est pas nécessaire de disposer de telles informations concernant les autres processus.

Le rétablissement de la topologie de l'infrastructure de communications est faite avec la participation du démon père du démon qui vient d'être relancé. Le démon qu'il relance se connecte à lui, et il lui transmet les points de contact de ses enfants, afin qu'il se connecte vers eux. Le rétablissement de l'infrastructure de communications se fait alors en inversant les connexions avec le niveau inférieur : elles sont sont pas faites du niveau inférieur vers le niveau supérieur, mais du niveau supérieur vers le niveau inférieur. Les nouvelles informations de connexion du processus ressuscité sont alors transmises vers le haut de l'arbre pour mettre à jour les informations détenues par les ancêtres du démon ressuscité.

Si le correspondance des processus MPI sur les démons de l'environnement d'exécution n'a pas été modifiée, le père du démon relancé peut relancer les processus victimes de la panne lui-même, en envoyant la commande correspondante. Sinon, comme dans le cas du protocole centralisé, le processus central effectue une diffusion pour lancer les processus à relancer à la manière de ce qui est fait pour le lancement initial de l'application, et les processus MPI victimes de la panne sont relancés, éventuellement à partir de points de reprise.

L'avantage de ce protocole en arbre, tenant compte de la topologie de l'environnement d'exécution et donc de l'infrastructure de communications, est qu'il permet de distribuer partiellement le service de réparation de l'infrastructure (même si l'attribution d'une nouvelle ressource demeure centralisée) et de préserver la topologie en faisant reprendre au nouveau démon exactement la même place que celle occupée par le démon victime de la panne, plutôt que de lancer tous les démons au niveau du démon central, dégradant ainsi la topologie. Ce protocole permet également de confiner cette panne et ses conséquences, en n'ayant aucun impact sur les processus de l'environnement d'exécution ne faisant pas partie de la même branche de l'arbre ou situés plus haut que le père du démon victime de la panne.

Rétablissement des communications de l'application Deux actions doivent être effectuées pour rétablir les communications entre les processus de l'application : les nouveaux processus doivent pouvoir communiquer avec le reste de l'application, et les processus survivants doivent pouvoir communiquer avec les processus relancés. En effet, contrairement à ce qui peut être fait avec certaines topologies (comme la topologie en arbre) dans l'environnement d'exécution, chaque processus de l'application doit pouvoir contacter directement chaque autre processus, et doit donc disposer des informations de communications de tous les autres processus.

Les informations de communication des processus de l'application font partie de l'état de l'environnement d'exécution : on cherche alors à les restaurer dans les processus qui viennent d'être relancés et à les corriger dans les processus survivants.

Les informations de communications des nouveaux processus sont donc diffusées auprès des processus survivants. Les nouveaux processus doivent également obtenir les informations de communications de tous les processus survivants. Ces informations sont disponibles dans les autres processus : il serait possible d'effectuer une opération de concaténation (semblable à un MPI_Gather) et de diffuser les informations de communications des nouveaux processus auprès des processus survivants. Cependant, pour des raisons de performances, nous avons choisi de garder une copie de l'ensemble des informations de communications de tous les processus de l'application à un endroit (le démon central, qui coordonne la tolérance aux pannes) et d'effectuer un envoi simple de l'ensemble des informations de connexion. Ces informations sont mises à jour avec les informations de connexion des nouveaux processus, et celles-ci sont diffusées entre tous les processus de l'application afin que les processus mettent à jour les données dont ils disposent localement. Pour résumer, la mise à jour se fait de la façon suivante :

- Envoi des informations de connexion des nouveaux processus par leur démon vers le démon central ;
- Diffusion de ces informations auprès des processus de l'application, qui mettent à jour leurs données de connexion locales ;
- Mise à jour des informations contenues dans le démon central ;
- Envoi de ces informations aux nouveaux processus.

4.5.2.4 Implémentation

L'implémentation de ce protocole a été faite dans OpenMPI, pour supporter les protocoles implémentés dans le PML-V. Sans activer le PML-V mais en utilisant les fonctionnalités de gestion d'erreur de la norme MPI 2.*, il permet également une gestion basique des erreurs au niveau de l'application MPI, dans la limite de ce qui est permis par l'état actuel de la norme.

Détection de panne La détection de pannes se fait au niveau des communications hors-bande entre les démons. L'infrastructure de communications formant un arbre couvrant, si un démon est victime d'une panne, un autre démon sera connecté à lui et le détectera.

La connexité de l'environnement d'exécution peut être perdue, par exemple dans le cas d'une topologie de l'infrastructure de communications en arbre, si un nœud tombe en panne, ses descendants sont déconnectés du reste du système ; le parent du processus victime de la panne détecte celle-ci et le rapporte aux niveaux supérieurs. Lors du rétablissement de l'état, le nouveau processus reprend sa place dans la topologie de l'infrastructure de communications, et la connexité est rétablie.

Dans le cas de pannes simultanées ou quasi-simultanées [4], il est possible que plusieurs processus successifs dans la topologie tombent en panne. Dans ce cas, il est nécessaire que la priorité soit donnée au processus qui est encore connecté au démon central de l'environnement d'exécution.

La détection d'erreur n'est pas appelée si la déconnexion du démon distant est considérée comme normale : par exemple en cas de terminaison de l'application, ou de réduction dynamique du nombre de processus.

Une fois l'erreur détectée, le composant chargé de la gestion des erreurs, *ERRMGR*, est appelé. C'est ce composant qui décide des actions à prendre en cas de défaillance d'un processus. Le composant par défaut, fourni avec OpenMPI, organise la terminaison de l'application. La tolérance aux pannes est permise dans l'environnement d'exécution par un nouveau ERRMGR appelé *FT* qui implémente le protocole présenté ci-avant. C'est ce composant qui organise la cicatrisation de l'environnement d'exécution, en redémarrant un

4. Ici, "*quasi-simultané*" désigne deux pannes successives où la deuxième a lieu avant le rétablissement de l'état suivant la première panne.

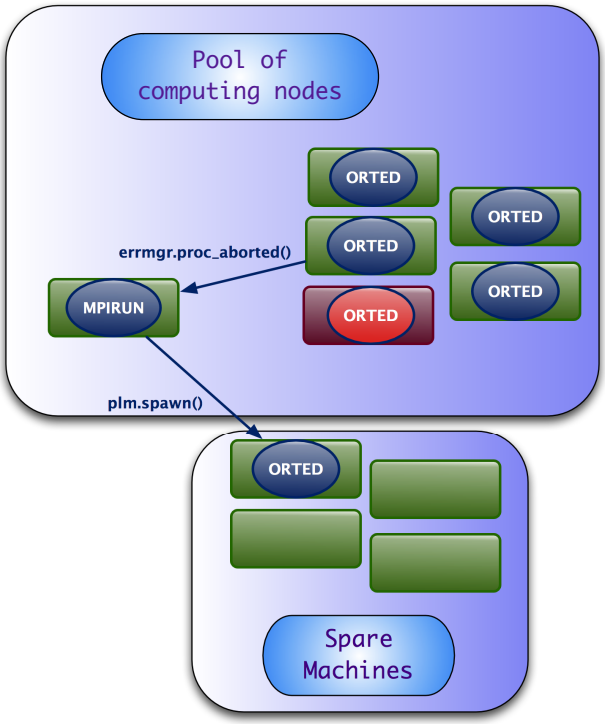


FIGURE *4.11 – Environnement d'exécution tolérant aux pannes : lancement d'un nouveau démon pour remplacer un mort*

nouveau démon. Il a également été nécessaire d'intervenir légèrement dans des composants d'autres frameworks afin de rendre possible la réintégration des nouveaux processus MPI, comme présenté dans ce qui suit.

Remplacement du processus Le gestionnaire d'erreur détermine quel démon a été victime d'une panne, et le relance sur un nouveau nœud. Ce nouveau nœud est sélectionné parmi un ensemble de nœuds non utilisés mais réservés à cette fin. Le relancement du démon victime de la panne est représenté figure 4.11.

Rétablissement de l'état Une fois le nouveau démon relancé, il est nécessaire de rétablir les services fournis à l'application MPI. Lorsqu'il est lancé, le démon transmet son information de contact au démon qui l'a lancé. La nouvelle information de contact du démon écrase l'ancienne dans l'infrastructure de communications de l'environnement d'exécution. Les communications étant rétablies, les services à l'application MPI peuvent être également rétablis.

Une fois l'infrastructure de communications rétablie, le nouveau démon reçoit l'ordre de lancer les processus MPI qui lui sont affectés. Il cherche la présence ou non d'une image du processus, sauvegardée précédemment par le mécanisme de sauvegarde d'état des processus. Si aucune n'est disponible, il redémarre le processus au début de son exécution. Sinon, il redémarre le processus dans l'état dans lequel il était lors de sa dernière prise d'état.

Les signaux et les entrées/sorties sont transmis par des communications hors-bande. Le rétablissement des communications dans l'environnement d'exécution et entre le processus MPI et son démon assurent le rétablissement de ce service.

Les communications MPI sont rétablies en transmettant les informations de contact des nouveaux processus aux autres processus, et les informations de contact de tous les autres processus aux nouveaux processus. Dans OpenMPI, les informations de contact sont transmises au moyen d'une opération collective appelée Allgather : chaque participant à l'opération collective reçoit, à l'issue de cette opération, la concaténation de toutes les informations transmises par les autres participants. Il s'agit d'une opération collective, qui doit ici être exécutée par un seul participant : on doit en fait exécuter une opération pseudo-collective. Les autres processus sont

impliqués dans cette opération de manière passive : ils reçoivent une mise à jour des informations de connexion des nouveaux processus, mais ne participent pas explicitement, par un envoi de message, à l'opération.

Ici, le démon central joue un rôle particulier. Il participe à l'opération uniquement pour garder une copie de ces informations concaténées. La version originale (non tolérante aux pannes) de l'environnement d'exécution d'OpenMPI ne conserve pas cette copie dans le démon central, cependant, même à très grande échelle, ces données restent de taille raisonnable (quelques méga-octets pour plusieurs centaines de milliers de processus), et permettent de n'effectuer qu'une communication point-à-point entre le démon central et le nouveau démon (qui, ensuite, met ces données à disposition de ses processus locaux), au lieu de nécessiter une communication collective de type concaténation entre tous les processus de l'application. Lorsque des processus sont relancés, ils envoient leurs informations de contact à ce processus central. Ces informations sont propagées à tous les processus de l'application. Les informations de contact concaténées sauvegardées dans le démon central sont modifiées afin de remplacer les informations de contact des processus victimes de la panne par les nouvelles informations de contact. L'ensemble est envoyé aux nouveaux processus. Il s'agit d'un pseudo-Allgather.

L'autre opération collective utilisée est la barrière. Une barrière assure une synchronisation entre ses participants : les participants ne sont autorisés à en sortir que si tous les participants y sont entrés. Tous les processus doivent effectuer cette synchronisation à la fin de l'initialisation afin de s'assurer qu'ils sont tous prêts à commencer l'application. Dans le cas d'un processus qui redémarre, l'environnement d'exécution n'a pas besoin de se re-synchroniser, puisque le démon qui redémarre et ses processus MPI cherchent, au contraire, à "rattraper" le reste de l'environnement d'exécution et de l'application. De la même manière que pour le pseudo-Allgather, une pseudo-barrière a dû être implémentée.

Il n'y a pas de relation d'ordre entre les envois d'informations de connexion, notamment entre la diffusion des informations de connexion des nouveaux processus et l'envoi de la concaténation de toutes les informations de connexion. Il est possible que l'on reçoive des informations de connexion déjà connues ou obsolètes, notamment en cas de pannes proches les unes des autres. Afin de ne pas créer de condition de course, un numéro d'époque est inséré au début du paquet contenant les informations de connexions envoyées. Ce numéro correspond à un comptage global de phases de redémarrage, et est incrémenté par le composant conservant une sauvegarde des informations de connexion à chaque pseudo-Allgather.

4.5.3 Performances du relancement

4.5.3.1 Modèle pour les performances

On peut modéliser le temps de la restauration de l'état de l'environnement d'exécution et la ré-initialisation de la bibliothèque de communications en utilisant les variables suivantes :

- Temps de détection de la panne : D
- Temps de relancement d'un nouveau démon : R
- Temps de connexion d'un processus au reste de l'infrastructure de communications : J
- Temps de relancement de l'application : B
- Temps du pseudo-Allgather : pA
- Temps de la pseudo-barrière : pB

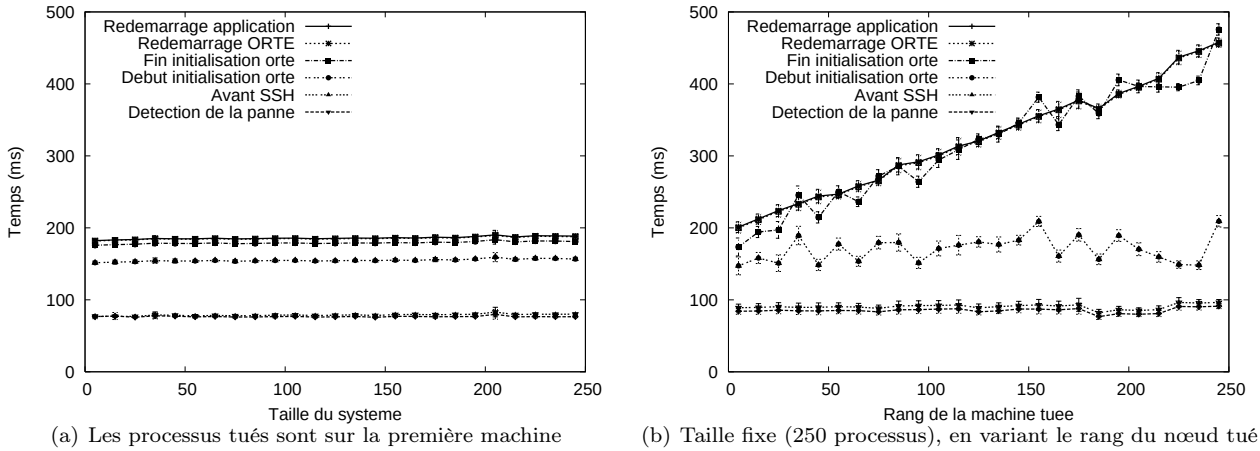
Le temps total de relancement d'une application est donc représenté par l'équation 4.3.

$$T_{rep} = D + R + J + B + pA + pB \tag{4.3}$$

La détection de la panne dépend du système utilisé : une bibliothèque de détection de panne, expiration des délais TCP... Ce temps dépend du système sur lequel l'application est exécutée.

Le relancement d'un nouveau démon est fait au moyen de deux parcours d'un tableau (tout le tableau des nœuds est parcouru deux fois, donc la complexité est en $O(N)$), et d'une connexion SSH ou RSH, notée S_{SSH}.

Le pseudo-Allgather est effectué par un envoi au démon central d'une quantité de données M_{mod}, le démon central diffuse cette information en un temps $M_{mod} \times T_{Bcast} = log_2(N) \times (M_{mod} \cdot \alpha + \beta)$, en désignant le nombre de nœuds par N, la latence du réseau par β et la bande passante par α. Dans cette notation, l'envoi d'un message a un coût de $data\_size \times \alpha + \beta$. On considère ici que l'algorithme de diffusion est un arbre binomial, nécessitant donc $log_2(N)$.



(a) Les processus tués sont sur la première machine

(b) Taille fixe (250 processus), en variant le rang du nœud tué

FIGURE *4.12 – Scalabilité du relancement d'un démon et de ses processus*

4.5.3.2 Mesures de performances

La figure 4.12 montre une mesure de la scalabilité du relancement des processus de l'environnement d'exécution et de l'application. Cette mesure a été effectuée sur la grappe de Grid'5000 GdX, en exécutant un processus MPI par machine.

L'application utilisée est une application MPI triviale effectuant un `MPI_Init` puis s'endormant pendant un temps long avant d'effectuer un `MPI_Finalize`. Un processus de l'application est tué pendant que tous sont en train de dormir. Le protocole de rétablissement de l'état de l'environnement d'exécution implémenté et étudié ici est celui utilisant un processus central pour orchestrer pour le relancement. On cherche ici à étudier sa scalabilité afin de déterminer s'il est suffisant pour passer à l'échelle.

Le relancement de l'environnement d'exécution a été instrumenté de façon à pouvoir isoler isoler les temps des étapes successives du relancement. Ainsi, on mesure le temps à la fin de l'injection de fautes (effectué par `ssh <machine> kill -9 -1`), et chaque démon relancé mesure le temps au début de son exécution (donnant ainsi le temps de déploiement du nouveau démon) et à la fin de la pseudo-initialisation des processus MPI. Les machines sont synchronisées par NTP avec une tolérance de décalage petite devant les valeurs mesurées. De plus, les expériences ont toutes été réalisées sur le même ensemble de machines, dans un même déploiement et en mélangeant les séries de mesures de façon équitable, de façon à ce que les déséquilibres éventuels soient les mêmes pour toutes les mesures.

Afin d'isoler les temps du processus de relancement, l'application a été relancée au début de son exécution, sans utiliser de points de reprise.

La figure 4.12(a) montre la scalabilité du relancement des processus de la *première* machine : le nombre de processus total augmente, et la machine tuée est toujours la première dans la liste des machines. Ainsi, du fait de l'algorithme de nommage des processus de l'univers OpenMPI, c'est sur elle que s'exécute le démon de rang 1 (le rang 0 étant le HNP) et le processus MPI de rang 0. La figure 4.12(b) montre le temps de relancement en fonction du rang du processus tué pour une taille du système fixe et égale à 250 processus. Un seul processus est exécuté sur chaque machine.

Même si les décalages entre les machines sont faibles grâce à la synchronisation fournie par NTP, nous avons effectué un maximum de mesures localement, au niveau du processus central qui orchestre le rétablissement de l'état. La première étape consiste à choisir le moment de la prise du temps. En effet, la panne d'une machine est simulée par l'envoi d'une commande `kill` en utilisant SSH. Nous avons donc tout d'abord mesuré la variabilité d'une connexion SSH sur la grappe utilisée. Sur une série de cent mesures sur chacune des machines utilisées, nous avons constaté une variabilité de l'ordre de 1%. Le temps d'effectuer cette connexion SSH est donc relativement stable : nous avons alors choisi d'effectuer la prise du temps initial juste avant l'envoi de la commande tuant les processus distants.

Détection de la panne On constate que le temps de détection de la panne est constant quelle que soit la taille du système et quel que soit le rang du processus tué.

Pour cette série de mesures nous avons détecté la panne au niveau socket : lorsque la commande `kill` tue des processus, l'environnement d'exécution hôte ferme les sockets de ces processus en respectant le protocole TCP. Le processus situé à l'autre bout de la connexion est donc averti de la fermeture de la socket. Cette injection de panne et la détection qu'elle permet ne sont pas des conditions totalement réalistes : dans le cas d'une défaillance survenant en conditions réelles, lorsque la machine tombe en panne elle n'a pas le temps de fermer proprement ses sockets. Il faut alors utiliser un détecteur de défaillances, comme le détecteur basé sur les battements de cœur [1] ou attendre l'expiration du délais TCP.

Cependant, cette mesure nous permet de considérer l'isolation du temps de rétablissement de l'environnement d'exécution et de relancement des processus victimes de la panne en montrant qu'il s'écoule un temps constant entre le moment où la panne est injectée et le moment où elle est détectée par le gestionnaire d'erreurs.

Relancement du démon de l'environnement d'exécution Le temps nécessaire pour arriver au moment où le système de lancement relance le démon victime de la panne est constant quelle que soit la taille du système et quel que soit le rang du processus tué. Il nécessite en réalité de parcourir de la liste des machines disponibles ; cependant cette opération locale est effectuée en un temps qui est petit devant les temps nécessaires pour effectuer les opérations de communications impliquées dans le protocole de relancement.

Le relancement du démon distant est suivi d'une initialisation du démon de l'environnement d'exécution. Ce temps est mesuré sur la machine distante, ce qui explique les variations constatées lorsque le rang de la machine tuée varie : les variations correspondent aux variations de date locale. On constate par ailleurs, la liste de machines n'ayant pas été modifiées d'une exécution à une autre, que cette variation est constante d'une exécution à une autre (faible écart type) et que le temps entre le début et la fin de cette initialisation ne subit pas de variations aussi fortes. Cette mesure étant toujours effectuée sur la même machine pour la figure 4.12(a), la date distante ne subit pas de décalage de synchronisation.

On constate que ce temps est constant lorsque la taille du système varie mais que le rang du démon tué reste le même (figure 4.12(a)) mais qu'elle augmente linéairement avec le rang du démon tué lorsque celui-ci varie mais que la taille du système ne varie pas(figure 4.12(b)).

La fin du redémarrage de l'environnement d'exécution est signifié au processus central par un *callback*, qui constitue donc une mesure locale.

Relancement et ré-intialisation des processus MPI On constate que le temps de relancement des processus de l'application et leur ré-initisalisation prennent un temps augmentant selon une faible pente lorsque la taille du système augmente. D'après le modèle défini en section 4.5.3.1, on tend à penser que cette augmentation est due à l'augmentation de la quantité de données à transférer dans le pseudo-Allgather pour rétablir l'état de l'environnement en lui transférant l'intégralité des informations de connexion des autres processus (la quantité de données à transmettre est une fonction linéaire du nombre de processus dans le système). Ce temps est constitué, comme vu dans la section 4.5.3.1, de l'établissement de la connexion du nouveau démon avec le reste de l'infrastructure de communications (temps constant), du relancement de l'application (dépendant de l'algorithme utilisé, ici une diffusion selon un arbre binomial, donc augmentant logarithmiquement avec le nombre de nœuds), du pseudo-Allgather (maximum entre une diffusion et la mise à jour des informations de contact de tous les processus et envoi au nouveau démon) et pseudo-barrière (temps constant). La mise à jour des informations de connexion et leur envoi en utilisant une communication point-à-point est donc la raison pour laquelle ce temps augmente légèrement quand la taille du système augmente.

On constate par ailleurs que le temps de relancement d'un processus de l'application même dans le pire cas (dernière machine) est significativement inférieur au temps de lancement total de l'application (environ deux secondes pour 250 nœuds d'après le chapitre 3). Outre le fait qu'elle permette de ne redémarrer que les processus victimes de la panne tout en laissant les autres processus poursuivre leur exécution, l'approche non-coordonnée permet donc à l'application d'arriver plus rapidement dans une situation où elle va pouvoir rétablir son état. Le support de l'environnement d'exécution propose donc une approche passant mieux à l'échelle qu'une approche coordonnée, et donc un support pour permettre de mettre en place des protocoles scalables de tolérance aux pannes au niveau de l'application.

4.6 Conclusion

On a vu dans ce chapitre le rôle de l'environnement d'exécution dans la tolérance aux pannes pour les applications MPI, en ajoutant à ses fonctionnalités de base un rôle de support du protocole utilisé au niveau de l'application MPI. Ce support consiste à permettre à l'application de retrouver son état après avoir subi une panne.

Les stratégies de retour sur points de reprise coordonnés nécessitent un support limité de l'environnement d'exécution, qui n'a pas besoin d'être lui-même tolérant aux pannes mais "conscient" de la tolérance aux pannes : il doit supporter l'enregistrement des points de reprise, la détection de fautes et le redémarrage de l'application sur un ensemble de points de reprise, mais il est finalisé et redéployé à chaque redémarrage de l'application.

Il a été montré que les stratégies coordonnées sont plus simples et ne complexifient pas le chemin critique des communications de l'application, et introduisent un surcoût inférieur aux stratégies non-coordonnées en l'absence de pannes. Ces protocoles offrent donc de meilleures performances sur des grappes de petite taille, et sont d'autant plus efficaces dans ce contexte qu'ils sont simples.

Cependant, les temps d'enregistrement des points de reprise et de redémarrage de l'application deviennent à grande échelle déraisonnables devant l'évolution du temps moyen avant défaillance. On doit alors se tourner vers des stratégies qui ne présentent pas de synchronisation globale, comme les protocoles non-coordonnés. Les protocoles basés sur de la journalisation de messages présentent un surcoût durant l'exécution sans pannes, mais engendrent une perte de calcul moindre lors du redémarrage après une panne, et surtout, ne nécessitent pas de synchronisation.

Les protocoles non-coordonnés nécessitent un support de l'environnement d'exécution plus actif que les protocoles coordonnés : il doit survivre aux fautes et continuer à fournir ses services au reste de l'application qui n'a pas été touché par la panne, tout en ce cicatrisant. En ce sens, il est lui-même tolérant aux pannes. L'environnement d'exécution ne peut pas suivre la même approche de retour sur points de reprises, et doit être auto-cicatrisant en utilisant un protocole de rétablissement d'état en reconstruisant le processus de l'environnement d'exécution victime de la panne.

En suivant cette approche, un environnement d'exécution tolérant aux pannes peut servir de support à des approches de la tolérance aux pannes au niveau applicatif (guidée par le programmeur de l'application), à l'image de FT-MPI [74], et des fonctionnalités de tolérance aux pannes prévues dans la norme MPI3. Cette famille d'approches propose au programmeur le choix de réparer l'infrastructure dans le sens où l'environnement d'exécution est réparé et les processus sont relancés, ou de conserver un nombre réduit de processus et continuer à fonctionner malgré cela.

Le protocole de rétablissement de l'état de l'environnement d'exécution est ici dit "auto-cicatrisant", mais d'autres approches pourraient être étudiées ; en particulier, l'auto-stabilisation [150, 71] peut apporter des algorithmes de construction et de maintien de topologies pour les réseaux de couches tels que les infrastructures de communications des environnements d'exécution [22, 23].

Le chapitre suivant présente un type particulier de systèmes à grande échelle : les grilles de calcul, formées de l'agrégation de plusieurs systèmes de taille plus modeste. Les conditions matérielles particulières liées à ces systèmes imposent un support spécifique de la part de l'environnement d'exécution, qui seront détaillées dans ce chapitre.

Chapitre 5

Intergiciel pour MPI sur les grilles

Un type particulier de système à grande échelle peut être obtenu en agrégeant des ressources distribuées géographiquement : il s'agit des grilles de calcul. En mettant en commun des ressources à plus petite échelle, on additionne leurs capacités de calcul et de stockage. À partir de plusieurs systèmes de taille modeste on peut ainsi obtenir un système à grande échelle.

Il existe deux principaux types de grilles de calcul. En premier lieu on trouve les fédérations de grappes, ou grappes de grappes, qui mettent en commun plusieurs grappes. En second lieu viennent les grilles de PC de type plateforme de calcul volontaire, constitués de machines individuelles complètement indépendantes les unes des autres, ou les grilles d'entreprises qui utilisent des ressources non utilisées en effectuant du vol de cycles. Les grilles étudiées ici sont des fédérations de grappes.

De tels systèmes supportent l'exécution d'applications sur plusieurs grappes à la fois, en les reliant par un réseau global et public (généralement le réseau Internet). Ce réseau n'est pas exclusif ; les machines et les grappes y accédant et étant accédées depuis l'Internet doivent être protégées par des dispositifs de sécurité comme des pare-feux ou des mécanismes de traduction d'adresses. Ce dernier dispositif est également souvent utilisé pour palier le manque d'adresses IP publiques.

La mise en place d'équipements de sécurité limite la connectivité au sein de la grille. C'est le but de cette politique de sécurité afin de protéger ses ressources, mais c'est également un obstacle à l'exécution d'applications distribuées sur la grille. En effet, celle-ci a, dans le cas général, besoin d'effectuer des communications entre ses processus. Or, les communications à travers la grille sont sensées être bloquées par les équipements de sécurité. De plus, chaque domaine d'administration définit sa propre politique de sécurité. Il peut donc y en avoir plusieurs cohabitant dans la même grille de calcul.

Pour pouvoir exploiter ces systèmes à grande échelle, il est donc nécessaire de permettre à l'application d'effectuer des communications entre les grappes, sur un réseau global partagé, sans abaisser le niveau de sécurité protégeant les grappes.

Les approches adoptées jusqu'à présent pour contourner les pare-feux consistent à ouvrir un passage dans ceux-ci et y faire passer les communications. Cette ouverture constitue une brèche dans la sécurité de la grappe : les pare-feux sont ouverts, certes partiellement, mais ils rendent alors le site vulnérable. Des ports ouverts dans un pare-feu impliquent par ailleurs que les applications utilisant ces ports disposent d'un mécanisme d'authentification permettant au serveur de reconnaître un client légitime et d'être capable de résister à une attaque venant de l'extérieur (débordements de tampons impossible, etc.). La mise en place de ces mécanismes de sécurité a un coût logiciel devant être pris en charge directement par l'intergiciel de communications et présentant un surcoût sur les performances.

Des machines utilisant des adresses privées peuvent être utilisées dans un calcul sur une grille en routant les messages par un démon de communications situé au niveau de la traduction d'adresse, et ayant accès au réseau public comme au réseau privé. Cette solution pose un problème de performances et nécessite d'avoir accès à la machine effectuant la traduction d'adresse. Une autre solution consiste à définir des règles statiques de traduction d'adresse, pour que des machines du réseau privé soient accessibles depuis l'extérieur en utilisant l'adresse du réseau et un port déterminé. Le routeur devient alors un point central de congestion et on retrouve le même problème de sécurité qu'en ouvrant des ports dans le pare-feu : les machines étant alors accessibles depuis l'extérieur, elles deviennent vulnérables et un mécanisme d'authentification et de sécurité doit être supporté par

l'intergiciel de communication.

Afin de supporter l'exécution d'applications distribuées sur les grilles, une pile logicielle a été conçue et implémentée en réponse à des problématiques identifiées et définies dans la première partie de ce chapitre. Le système QosCosGrid est décrit dans la section 5.2. Il s'agit d'une pile logicielle complète, des systèmes de réservation à grain fin aux bibliothèques de communication s'interfaçant avec les applications. La bibliothèque de communications utilisée par QosCosGrid est décrite section 5.4 et ses performances sont analysées section 5.5.

La bibliothèque de communications utilisée dans le système QosCosGrid permet d'exécuter une application distribuée communiquant par passage de messages sur une grille de calcul, malgré la présence d'équipements de sécurité (pare-feux, traduction d'adresses). Plusieurs techniques d'interconnexion sont proposées, certaines ne nécessitant pas de configuration particulière risquant de compromettre la sécurité des grappes. Cette bibliothèque repose sur un environnement d'exécution étendu par un ensemble de services de grille formant une infrastructure répartie et permettant d'utiliser des techniques avancées d'établissement de connexions.

5.1 Applications parallèles pour grilles de calcul

Les grilles considérées ici sont de type "fédération de grappes", comme illustré par la figure 5.1.

Une grille de calcul est définie dans [82] comme étant un système remplissant la réunion des trois conditions nécessaires suivantes :

– "*Coordonner des ressources qui ne sont pas sujettes à un système centralisé...*
– *... en utilisant des protocoles et des interfaces normalisés et généralistes...*
– *... pour fournir une qualité de service non triviale.*"

La coordination des ressources constituant la grille est effectuée par un *intergiciel de grille*. Les problématiques liées à cette coordination sont liées à la distribution des ressources.

Si des outils spécifiques existent pour utiliser les grilles comme Globus [83], ce sont avant tout des interfaces de bas niveau. On parle parfois de "systèmes d'exploitations de grilles", bien qu'il ne s'agisse pas à proprement parler de systèmes d'exploitation mais d'outils de bas niveau pour l'exploitation de la grille. Les outils de programmation d'applications parallèles doivent se situer au niveau de la couche supérieure à ces outils.

Il existe plusieurs techniques de programmation pour programmer des applications parallèles sur les grilles de calcul. Le modèle d'appels à des procédures distantes (RPC [153]) est le premier a avoir été utilisé pour programmer des applications sur des grilles de calcul. Des adaptations de cette norme ont été faites pour la grille, comme GridRPC [129], pour NetSolve [45, 18, 164] ou Ninf-G [156, 130, 157] pour Globus 4. XML-RPC [1] est une implémentation conçue spécialement pour être utilisée sur Internet, utilisant le protocole HTTP pour ses communications et encapsulant les données à transmettre (arguments, valeurs de retour) dans du XML.

Les services de grille suivent également le modèle RPC. Ils utilisent la plupart du temps le protocole de communications SOAP [101], successeur de XML-RPC. SOAP peut utiliser le protocole HTTP(S) ou SMTP, ce qui le rend particulièrement adapté aux systèmes communiquant sur Internet. Axis [2] est un moteur de services web et de services de grilles (un service de grille est un service web à état) utilisant le protocole SOAP utilisable avec Globus. OGSA-DAI [8] est un moteur de services de grilles orienté vers l'accès et le traitement des données sur les grilles et pouvant être couplé avec OGSA-DQP [4], un processeur de requêtes distribuées.

Une autre façon de programmer des applications distribuées sur des grilles consiste à utiliser des flux de données (*workflows*). Condor [120, 158] est un système de parallélisation de tâches décrites par le flux de données de type "sac de tâches". Il peut être utilisé avec Globus (Congor-G [159]) en s'interfaçant avec son gestionnaire de ressources pour déployer les processus, et présente des possibilités de regroupement de tâches Condor s'exécutant sur différentes grappes au moyen d'un entremetteur. DAGman est un outil de Condor. C'est un ordonnanceur permettant d'optimiser l'exécution d'une application Condor en tenant compte des dépendances de données entre les tâches.

Le modèle de programmation d'applications sur systèmes à mémoire distribuée communiquant par passage de messages explicites, qui est le modèle suivi par MPI, est aussi utilisable sur les grilles de calcul. Il présente l'avantage d'être portable et utilisé par une large communauté. Sa portabilité permet de réutiliser des applications initialement écrites pour être exécutées sur des grappes. De nombreuses bibliothèques scientifiques sont

1. Internet Remote Procedure Call : XML-RPC. `http://www.xmlrpc.com/spec`
2. Web services - Apache Axis. `http://ws.apache.org/axis`

disponibles et l'expérience acquise par les développeurs d'applications scientifiques en font une solution attractive pour développer des applications parallèles pour les grilles en utilisant la même technologie que pour les grappes. Cependant, les environnements de programmation et d'exécution d'applications MPI pour les grilles doivent répondre à des contraintes spécifiques et nouvellement introduites par les grilles de calcul.

5.1.1 MPI sur les grilles : état de l'art

Étant interconnectées par un réseau public (le réseau Internet), les grilles nécessitent la mise en place de moyens de sécurité afin, d'une part, de les protéger des intrusions venant de l'extérieur, et d'autre part, de protéger les données circulant sur le réseau public. L'infrastructure GSI [162], liée à Globus [83], fournit une authentification entre les ressources de la grille, un service d'authentification des utilisateurs basé sur des certificats signés par une autorité de certification et/ou une authentification par proxy et une correspondance entre l'utilisateur de la grille et un utilisateur local sur les ressources, ainsi que des possibilités de cryptage des communications. Les intergiciels de calcul existants tels que MPICH-G2 [111], PACX-MPI [87, 86] ou GridMPI [127, 155] sont compatibles avec l'utilisation d'équipements de sécurité tels que des pare-feux mais nécessitent toutefois que des ports soient ouverts pour pouvoir communiquer. PadicoTM [70] offre une couche de communications pour des applications pour les grilles utilisant en même temps un paradigme de programmation parallèle (avec MPI) et distribuée (avec CORBA). Il propose une technique d'interconnexion permettant de passer les pare-feux même entièrement fermés reposant sur la technique du TCP splicing [69] qui sera présentée plus en détails dans la section 5.4.3.5.

L'hétérogénéité des ressources de calcul et de communication pose des problèmes spécifiques auxquels les intergiciels de communication doivent faire face. Plusieurs types de réseaux peuvent être rencontrés, des réseaux rapides, à portée locale, aux réseaux longue distance. Les intergiciels de communication doivent être capables de supporter plusieurs protocoles, et éventuellement, de router les communications à travers plusieurs protocoles. Une bibliothèque s'est spécialisée dans les communications sur réseaux hétérogènes : MPICH-Madeleine [12]. On peut aussi citer GridMPI [127, 155], qui vise à optimiser les communications traversant des réseaux locaux rapides et des réseaux globaux. Sans être particulièrement orientée vers les grilles, OpenMPI [85] est une bibliothèque de communications multi-protocoles et est capable de supporter une exécution sur plusieurs types de réseaux à la fois. PACX-MPI [87, 86] établit une sorte de "lien" sur un réseau longue distance entre les réseaux locaux ; le routage entre deux réseaux locaux se font en passant par des démons de communications. En ce sens, on peut dire qu'il est multi-protocoles. MPICH [99] n'est pas multi-protocoles : la bibliothèque doit être compilée pour un canal déterminé [97] et ne peut pas en utiliser plusieurs à la fois. MPICH-G [80] utilise l'interface de communications Nexus [81], qui supporte les communications à travers différents protocoles. Cette bibliothèque spécialisée pour les grilles Globus [83] a depuis été remplacée par MPICH-G2 [111] et n'utilise plus Nexus mais un canal de communication spécifique, *globus2*. Celui-ci supporte les réseaux TCP et les réseaux locaux propriétaires fourni avec les machines, si la bibliothèque de communication correspondante est installée sur la machine.

L'hétérogénéité des ressources peut avoir pour conséquence des différences de représentations des données à travers la grille : représentation 32 ou 64 bits, petit-boutiste ou gros-boutiste, différences de précision pour les nombres a virgule flottante, et parfois des décalages moins triviaux, comme lorsque des processeurs Cell sont impliqués. PACX-MPI [87, 86] traduit toutes les données transférées sur le réseau global dans un format tiers appelé XDR (eXternal Data Representation). Nexus [81] pour MPICH-G [80] et Globus [83] pour MPICH-G2 [111] se chargent de cette traduction. GridMPI [3] [127, 155] ne supporte pas toutes les architectures hétérogènes en n'effectuant pas ces opérations de traduction non triviales. OpenMPI [85] supporte toutes sortes d'hétérogénéités, même les décalages non triviaux entre processeurs Cell et autres architectures. MPICH [99] et MPICH2 [37] ne supportent pas l'hétérogénéité des représentations des données.

L'environnement de programmation et d'exécution proposé ici, appelé QCG-OMPI, permet d'interconnecter des processus situés dans des domaines d'administration différents en utilisant des techniques d'interconnexion avancées permettant de passer les pare-feux et les traductions d'adresses. Plusieurs techniques sont disponibles, certaines permettant d'exploiter la configuration particulière de l'environnement, d'autres fonctionnant quelle que soit la configuration des équipements de sécurité. La bibliothèque de communication permet également de

3. GridMPI : `http://www.gridmpi.org`

s'adapter à l'hétérogénéité des ressources, en cas de besoin, en utilisant un format commun de représentation des données si deux processus ayant des architectures incompatibles doivent communiquer ensemble.

5.1.2 Définitions

L'*intergiciel de gestion et d'utilisation de grille* est en charge d'exploiter efficacement les ressources de la grille en assurant leur coordination et leur utilisation. J'ai participé à la conception de l'architecture globale du système QosCosGrid. Chaque composant a été implémenté et mis en place par les équipes en charge de la fonctionnalité correspondante.

Son architecture repose sur la notion de *domaine d'administration*. Un domaine d'administration est un ensemble de ressources ayant en commun une même politique d'administration, des règles de sécurité communes et des utilisateurs. Typiquement, une université est un domaine d'administration : les ressources sont administrées par une seule et même équipe d'administrateurs, elles sont reliées au réseau global en passant par une passerelle unique. Un dispositif de sécurité (pare-feu, traduction d'adresses publiques et privées, système d'authentification) protège toutes les ressources comme un seul ensemble et des utilisateurs y sont rattachés.

Les ressources de calcul sont assemblées en *grappes*. Une grappe peut être un ensemble de machines de type PC reliées par un réseau local, ou une machine massivement parallèle. On peut trouver plusieurs grappes par domaine d'administration, mais une grappe ne peut pas faire partie de plusieurs domaines d'administration. On peut également rassembler plusieurs grappes en les reliant par un réseau local. Par exemple, on peut relier par un réseau local de type Ethernet plusieurs grappes disposant d'un réseau rapide, et/ou plusieurs machines massivement parallèles. On peut aussi avoir une hiérarchie au sein d'un même domaine d'administration : deux grappes situées dans une même salle seront généralement interconnectées par un réseau plus rapide qu'un réseau reliant ces deux grappes à une troisième, située dans un autre bâtiment.

Chaque grappe dispose en outre d'un *système de réservation de ressources* local. Il s'agit d'un système coordonnant l'utilisation des ressources (à l'échelle d'une machine entière ou d'un cœur) et enregistrant des réservations et en autorisant leur utilisation uniquement pendant la durée de la réservation.

Les *ressources de calcul* sont composées d'une mémoire locale, d'un certain nombre d'unités de calcul, d'au moins une interface réseau et éventuellement d'un disque dur local. Les unités de calcul comptent au moins un CPU par machine. Ce CPU peut être composé de plusieurs cœurs. Les machines peuvent disposer de plusieurs CPU (SMP), et éventuellement d'accélérateurs (GPU, FPGA, processeurs Cell).

Une *tâche* est constituée d'un ensemble de processus s'exécutant sur un ensemble de ressources du système. Une numéro de tâche unique lui est attribué par le système de réservation de ressources et elle est rattachée à un utilisateur.

Un *utilisateur* peut accéder à la grille à distance. C'est lui qui soumet une tâche au système et en collecte le résultat. Le *programmeur*, quant à lui écrit le programme et ses fichiers compagnons.

Un exemple de grille est représenté figure 5.1. Le système est composé de trois grappes dont une grappe de PC. Un utilisateur accède au système par l'intermédiaire de sa machine locale.

5.1.3 Quasi-opportunisme

Un intérêt des grilles de calcul est de permettre l'agrégation, éventuellement temporaire, de ressources de calcul et de stockage. Les utilisateurs ont un besoin ponctuel en ressources pour exécuter les applications cibles. Le caractère ponctuel de ce besoin fait qu'il ne justifie pas l'investissement dans une grosse machine, une grappe de taille relativement modeste étant suffisante la plupart du temps.

Lorsque des ressources plus importantes s'avèrent nécessaires, plusieurs institutions partenaires mettent en commun leurs ressources et forment une grille sur laquelle s'exécutent la ou les applications demandeuses de ressources dépassant les capacités d'une grappe locale.

Les utilisateurs étant rattachés au domaine d'administration de l'institution dont ils font partie, il est possible de mettre en place un système de comptage du temps-machine utilisé auprès des autres institutions et d'établir un système économique.

En-dehors du contexte de quasi-opportunisme, le système QosCosGrid peut être utilisé pour les grilles de calcul en général. Il permet d'exploiter un ensemble de ressources, regroupées en grappes, comme un tout.

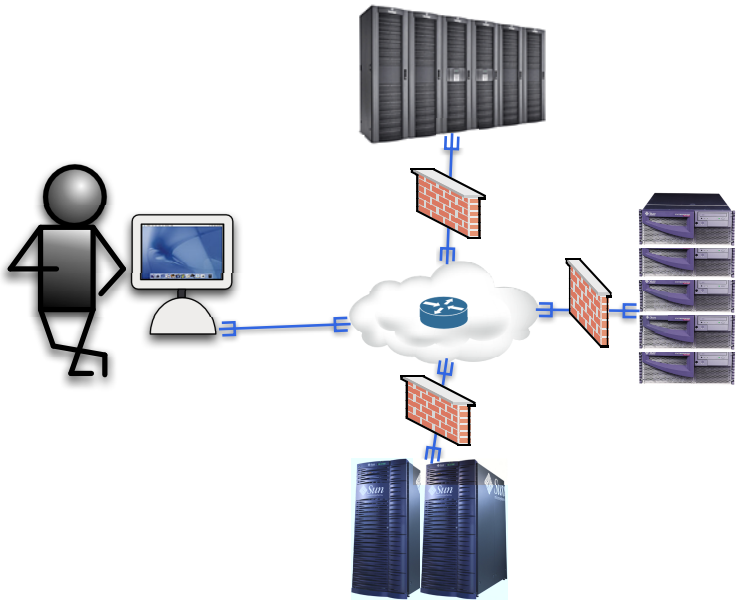


FIGURE *5.1 – Un exemple de grille de calcul : plusieurs grappes et/ou machines massivement parallèles sont interconnectées par un réseau global (Internet), donnant accès aux utilisateurs à une agrégation de ressources.*

5.1.4 Problématique de connectivité et sécurité

L'intergiciel de gestion et d'utilisation de la grille a pour but de fournir à l'utilisateur une impression proche de celle qu'il a lorsqu'il dispose d'une grappe de calcul. Les différentes couches du système QosCosGrid doivent donc faire face à plusieurs problématiques décrites en introduction de cette section et dont les suivantes seront traitées dans ce chapitre.

Les domaines d'administration sont protégés par des politiques de sécurité visant à interdire les intrusions non autorisées. On les protège alors par des pare-feu. Une traduction d'adresse entre l'adresse publique du domaine d'administration et les adresses privées utilisées à l'intérieur du domaine peut également être utilisée pour se protéger des intrusions et pour limiter le nombre d'adresses IP publiques nécessaires.

Cependant, une application s'exécutant sur plusieurs domaines d'administration a besoin de communiquer entre ces domaines. Il est donc nécessaire de fournir une bibliothèque de communications capable de communiquer de manière transparente entre les domaines d'administration en traversant les dispositifs de sécurité, sans abaisser le niveau de protection.

Pour cela, nous avons développé un ensemble de techniques de connectivité permettant d'établir des connexions entre des processus protégés par un dispositif de sécurité. Ces techniques sont présentées dans la section 5.4.2 et permettent aux processus de communiquer les uns avec les autres de manière transparente.

5.2 Architecture

Le système QosCosGrid est composé de composants centraux servant à l'orchestration de la grille et à la synchronisation de composants locaux présents dans chaque domaine d'administration. L'architecture du système est schématisée figure 5.2. Elle est présentée dans [43, 115, 112, 113].

5.2.1 Composants centraux de QosCosGrid

5.2.1.1 Grid Security Infrastructure (GSI)

L'infrastructure de sécurité de la grille (ou GSI) est chargée d'assurer la sécurité sur la grille. C'est le niveau le plus bas du système d'authentification et de contrôle d'accès. Elle gère les autorisations d'accès aux services des

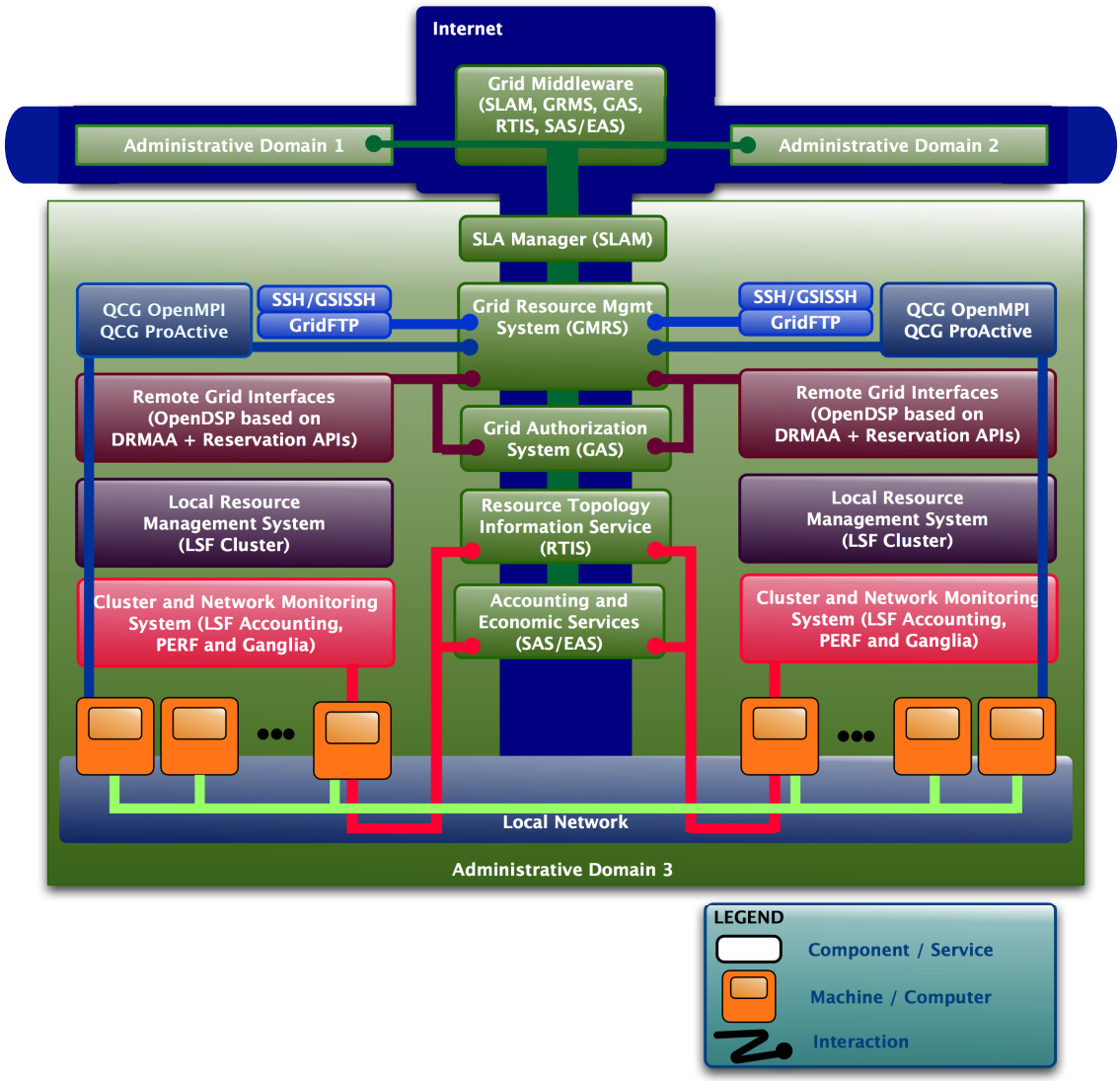


FIGURE 5.2 – *L'architecture QCG dans son ensemble, sur trois domaines d'administration (zoom sur le domaine numéro 3). Les machines, en orange, sont reliées entre elles par un réseau local connecté à Internet.*

utilisateurs et maintient des données telles que le rattachement des utilisateurs aux domaines d'administration.

5.2.1.2 Gridge Authorization Service (GAS)

Le système de gestion des autorisations repose sur GSI pour gérer les données d'autorisation entre domaines d'administration. Il est présent a la fois au niveau local (dans chaque grappe) et au niveau de la grille. Ainsi un utilisateur d'un domaine d'administration peut avoir accès à la grappe de son domaine d'administration sans avoir accès à toute la grille. Il sert de frontale d'accès à la base de données définissant la politique de collaboration entre domaines d'administration et permet de définir des politiques de contrôle d'accès à gros grain.

Le GAS et le GSI répondent à la problématique de sécurité et de gestion des authentifications au sein de la grille.

5.2.1.3 Service Level Agreement Manager (SLAM)

Le système de négociation et d'accord orchestre la négociation entre les domaines d'administration (fournisseurs de ressources) et les applications à ordonnancer (consommateurs de ressources). L'accord décrit le temps alloué et la qualité de service en termes de ressources, de topologie et de communications ainsi que la correspondance entre les processus de la tâche et les ressources physiques.

Le SLAM a pour but d'ordonnancer les processus des tâches de manière cohérente sur les ressources disponibles.

5.2.1.4 Resource Topology Information Service (RTIS)

Le service d'information sur la topologie des ressources est un système d'information contenant les données décrivant les ressources en termes de performances de communications, de quantité de mémoire disponible, ou toute autre mesure définissable. Les informations sont collectées et mises à jour à intervalle régulier par un système de mesure basé sur les services locaux de gestion des ressources. C'est ce système d'information qui est consulté par l'ordonnanceur afin de fournir un ensemble de ressources correspondant à la demande de l'utilisateur.

5.2.1.5 Gridge Resource Management System (GRMS)

Le meta-ordonnanceur de grille est chargé d'ordonnancer les tâches sur les ressources disponibles en fonction des données de budget données après négociation par le SLAM, en fonction des caractéristiques requises par l'utilisateur et des caractéristiques des ressources disponibles auprès du RTIS. Le meta-ordonnanceur de grille utilisé ici est le Gridge Resource Management System [114].

Le GRMS, le RTIS et le SLAM sont chargés de l'ordonnancement des tâches en fonction des requêtes formulées par les utilisateurs de la grille. Grâce aux informations fournies par le RTIS, le GRMS établit un plan d'ordonnancement en fonction de la négociation effectuée par le SLAM.

5.2.1.6 Accounting and Economic Services (SAS/EAS)

Les services économiques sont chargés de faire la facturation des ressources utilisées par les utilisateurs auprès des autres domaines d'administration, en fonction de la quantité de ressources utilisées, des caractéristiques demandées et de la qualité de service garantie.

Les services économiques répondent à une problématique de facturation du temps de calcul sur des ressources extérieures au domaine d'administration auquel l'utilisateur est rattaché. Les différents domaines d'administration impliqués dans une grille ayant une relation collaborative, il peut être nécessaire de comptabiliser le temps de calcul utilisés par les utilisateurs sur des domaines d'administrations qui ne sont pas celui auquel ils appartiennent. Ce temps peut alors être facturé au domaine d'administration de rattachement de l'utilisateur.

5.2.2 Composants locaux

Les composants locaux sont présents dans chaque grappe. Ils gèrent les ressources disponibles dans leur grappe uniquement, en étant éventuellement coordonnés à un niveau supérieur par les services centraux.

5.2.2.1 Monitoring

Chaque grappe dispose d'un système de surveillance des ressources au niveau des machines et du réseau. Les solutions disponibles pour QosCosGrid sont le système de monitoring intégré à LSF [4] et Ganglia [125].

Le monitoring est utilisé par les outils d'ordonnancement afin d'évaluer régulièrement l'état des ressources disponibles. Par exemple, la latence et la bande passante sont évaluées à intervalles réguliers afin de maintenir à jour le système d'informations contenant les informations sur les ressources disponibles (RTIS). Ces données peuvent fluctuer dans le temps, principalement en raison de la concurrence entre les applications. Le RTIS a alors connaissance des performances réelles à un instant donné des liens réseaux entre les ressources de calcul et

4. LSF : Load Sharing Facility. `http://www.platform.com/Products/platform-lsf`

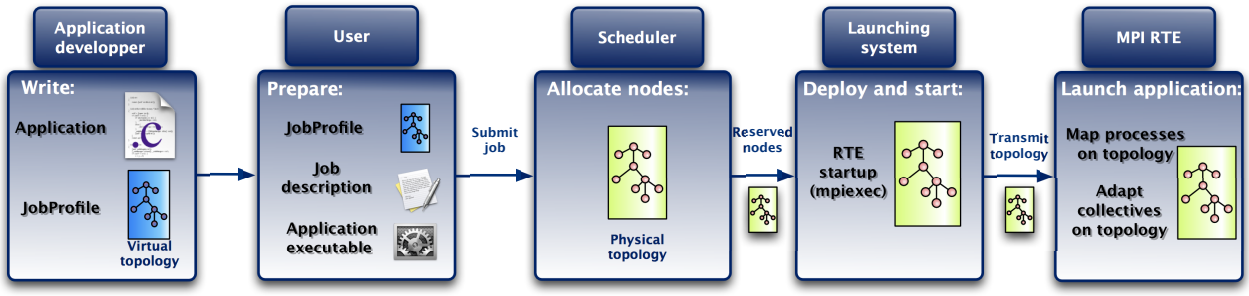


FIGURE *5.3 – La chaîne de vie d'une tâche QCG, de la conception par le programmeur à l'exécution.*

peut les transmettre au méta-ordonnanceur de grille, qui peut établir des plans d'ordonnancement des processus en tenant compte de la bande passante et de la latence disponibles.

D'autres métriques peut être définies, utilisant les informations données par les machines et les capteurs dont elles disposent. Par exemple, il peut être décidé de ne pas réserver de machines dont la température CPU est trop élevée, à cause d'un problème matériel au niveau du refroidissement ou d'un calcul précédent qui l'aurait fait chauffer. Une règle doit alors être définie auprès du système de monitoring local et prise en compte par le RTIS et le GRMS, mais elle n'a pas à être définie sur toutes les grappes.

5.2.2.2 Système de gestion de ressources local

Chaque grappe dispose d'un système de gestion de ressources permettant de réserver des ressources en accès exclusif. Ce sont les systèmes de gestion de ressources locaux qui effectuent les réservations demandées par le meta-ordonnanceur de grille. Ils doivent également transmettre l'état de leurs réservations au meta-ordonnanceur de grille afin de préserver la cohérence de la vue que celui-ci a de l'état du système.

5.2.2.3 Interfaces de réservation pour la grille

Sur chaque grappe un système de réservation de ressources à distance permet d'effectuer des réservations sur le système de gestion de ressources local afin de permettre de coordonner les réservations entre les grappes. Le système de meta-réservation utilisé ici est OpenDSP [161]. Il coordonne les réservations émises par les systèmes de réservations locaux.

5.2.2.4 Environnement parallèles

Les applications disposent d'environnements de programmation et d'exécution parallèles afin de pouvoir s'exécuter en parallèle sur la grille. Ces bibliothèques sont présentes dans toutes les grappes et répondent à des problématiques spécifiques aux grilles de calcul, en termes de communications et de fonctionnalités.

C'est sur ce composant que j'ai spécifiquement concentré mon travail. L'environnement parallèle doit permettre aux applications de communiquer sur la grille, malgré la présence éventuelle d'équipements de sécurité. QosCosGrid propose deux environnements de programmation : QCG-ProActive, une bibliothèque de communication par passage de messages communiquant par RMI, basée sur la bibliothèque pour applications Java ProActive [44], et QCG-OMPI, basée sur l'implémentation de la norme MPI OpenMPI [85]. QCG-ProActive ne bénéficie pas des techniques de connectivité avancées dont bénéficie QCG-OMPI, sur lequel ont été concentrés les efforts de recherche. J'ai contribué uniquement à QCG-OMPI. Son architecture est décrite dans la section 5.4.

5.3 Vie d'une tâche

Le système QosCosGrid prend en compte le cycle de vie d'une application dans son ensemble, de la conception à la fin de l'exécution. Mon travail a porté sur la conception de ce cycle de vie, décrit dans [43].

Le cycle de vie de l'application, de son développement à son exécution, est représenté figure 5.3.

5.3.1 Soumission

Le système de soumission joue un rôle clé dans la chaîne de vie d'une application QosCosGrid car c'est à lui qu'est confiée la responsabilité de l'utilisation d'un ensemble de ressources compatibles avec l'application et ses schémas de calcul et de communication.

Une tâche est soumise au système en lui fournissant le fichier exécutable de l'application et le fichier de description des ressources demandées. Cette description est plus complexe qu'une simple quantité de ressources car elle inclut des spécifications sur les ressources elles-mêmes, comme la quantité de mémoire disponible ou les caractéristiques des liens réseaux interconnectant les machines.

Les emplacements des fichiers de données d'entrées, les fichiers de sortie des résultats, les entrées/sorties standards, la qualité de service demandée (fidélité par rapport aux ressources demandées, ressources supplémentaires en cas de panne) ainsi des caractéristiques sur l'environnement d'exécution peuvent être spécifiés lors de la soumission.

5.3.2 Réservation

Au regard de la demande en ressources et du budget prévu par le système de facturation, le meta-ordonnanceur sélectionne et réserve des ressources pour la tâche. Le meta-ordonnanceur orchestre les réservations au niveau de la grille et transmet aux systèmes de réservations locaux qui effectuent les réservations à grain fin dans les grappes.

5.3.3 Déploiement

Les systèmes de déploiement locaux se chargent du déploiement et du lancement de l'application en étant coordonnés par le système de déploiement au niveau de la grille (OpenDSP). Si besoin, celui-ci peut se charger de transférer des fichiers comme l'exécutable ou des fichiers de données d'entrée.

5.3.4 Exécution

L'exécution se déroule dans un environnement d'exécution parallèle adapté aux problématiques de grilles et offrant la possibilité d'interagir avec le système de soumission. Il est possible pour l'environnement d'exécution de l'application parallèle d'obtenir des informations de la part du système de soumission, notamment sur les ressources allouées par le meta-ordonnanceur.

5.3.5 Finalisation

Le système de déploiement au niveau de la grille s'assure de la terminaison de l'application dans son ensemble, de la récupération des données de sortie et de la restauration de l'environnement des machines utilisées (nettoyage des fichiers temporaires créés). Il peut aussi forcer la finalisation de l'application à la fin du temps qui lui a été imparti afin d'assurer le respect des contraintes de temps.

5.4 Connectivité inter-grappes

Les applications s'exécutant sur plusieurs grappes distribuées sur plusieurs domaines d'administration doivent pouvoir s'exécuter et communiquer sur toutes les ressources, malgré la présence d'équipements de sécurité tels que des pare-feux et/ou des traductions d'adresses (NAT, NAPT). Pour cela, on peut utiliser des techniques de connectivité simples ou nécessitant des manipulations du protocole TCP. Les techniques utilisées ici sont issues de PVC [146], un système de grilles instantanées permettant d'utiliser des ressources distribuées géographiquement comme une seule grappe.

La bibliothèque de communication utilisée ici est basée sur OpenMPI [85] et appelée QCG-OMPI. Elle est présentée dans [61, 65] et ses performances ont été évaluées dans [64, 60, 63, 115].

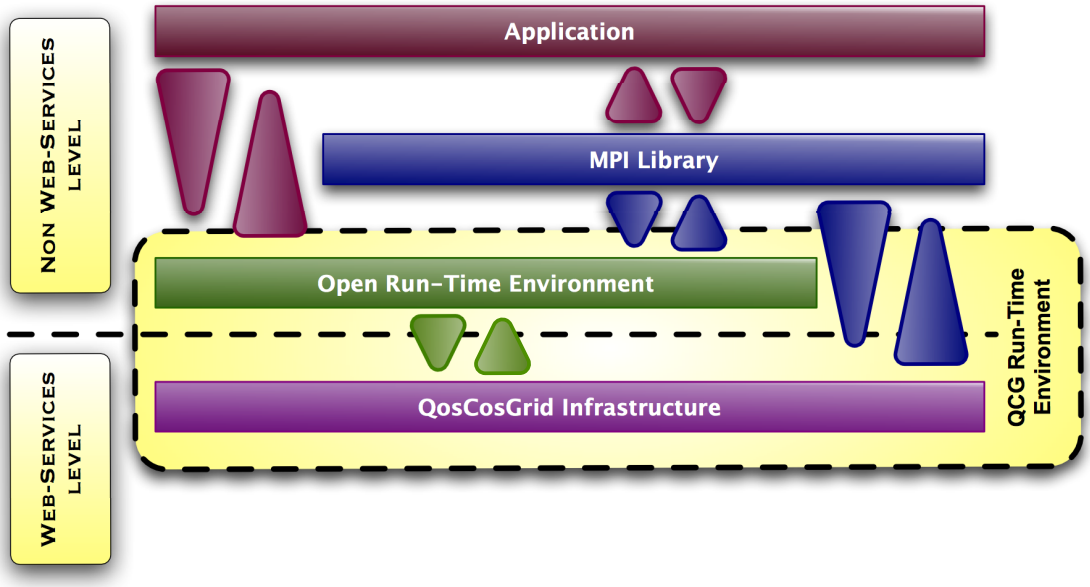


FIGURE 5.4 – *Architecture de la bibliothèque de programmation et d'exécution parallèles QCG-OMPI*

5.4.1 Extension de l'environnement d'exécution

L'établissement des communications sollicitant un support de l'environnement d'exécution, les services d'interconnexion avancés de QCG-OMPI sont rassemblés dans une extension de l'environnement d'exécution d'OpenMPI. L'architecture en couches de QCG-OMPI est présentée sur le schéma 5.4.

En jaune, entouré par une ligne pointillée, l'environnement d'exécution peut être découpé en deux parties distinctes : le niveau présent sous forme de services de grilles, et le niveau présent sous forme non-grille.

L'environnement d'exécution fournit des services à la couches MPI (routines de communications) et à l'application (lancement, informations sous forme d'attributs). Enfin, l'environnement d'exécution ayant besoin de services de communications pour les communications hors-bande, le niveau non-grille fait également appel au niveau grille pour établir ses connexions.

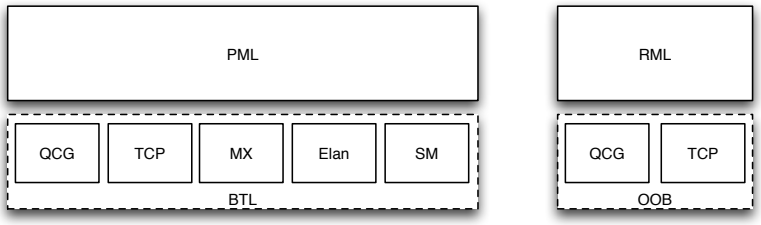
Au niveau des communications MPI (BTL) et internes (hors-bande, ou OOB, disponible pour réseaux TCP uniquement), QCG-OMPI dispose de pilotes spécifiques lui permettant de tirer parti de l'environnement d'exécution étendu (figure 5.5).

Les communications hors-bande sont disponibles dans OpenMPI uniquement pour les réseaux TCP. Un seul pilote OOB est utilisé à la fois.

Les communications utilisées par les routines de communications MPI sont implémentées pour plusieurs types de réseaux dans les BTL. Les processus d'une application peuvent utiliser plusieurs réseaux à la fois pour communiquer entre eux. À la fin de l'initialisation des modules BTL pour chaque interface réseau disponible, les processus effectuent une opération collective appelée *modex* (module exchange) où chacun met à la disposition des autres les moyens disponibles pour le contacter. Il s'agit typiquement pour les réseaux TCP d'un couple formé d'une adresse IP et d'un port sur lequel une socket écoute les connexions entrantes.

Le pilote BTL QCG ne prend pas part au modex, mais si d'autres interfaces réseau sont disponibles, il est effectué avec leurs identifiants de connexion. Les autres réseaux utilisables sont des réseaux locaux (Myrinet, Infiniband, Quadrix...), ne nécessitant pas l'utilisation de techniques de connectivité inter-grappes. De plus, ils sont prioritaires dans l'ordre d'utilisation des interfaces réseaux disponibles. Par conséquent, si un réseau local est disponible entre deux processus, QCG-OMPI les fera communiquer à travers ce réseau local. Sinon, le pilote QCG sera utilisé.

Le pilote QCG est un pilote de communications sur TCP faisant appel aux services de grilles de l'environnement d'exécution étendu dans l'établissement des connexions entre les processus. Les connexions entre les



(a) Le pilote de communication QCG dans l'architecture de communications d'OpenMPI.

(b) Le pilote de communication QCG dans l'architecture de communications d'ORTE.

FIGURE *5.5 – Les pilotes de communication QCG, permettant de tirer parti de l'infrastructure QCG.*

processus sont établies à la demande : la première fois qu'un processus doit envoyer un message à un autre processus, il effectue un `connect()` vers ce processus, qui, de son côté, effectue un `accept()` sur les connexions arrivant sur sa socket d'écoute. Les informations de connexion du processus distant (adresse IP et port sur lequel se connecter) sont obtenues à la demande, durant l'exécution, juste avant d'effectuer le `connect()`. Pour les obtenir, le processus qui doit effectuer le `connect()` fait un appel à l'infrastructure de grille au moyen d'un appel à une fonction distante d'un service de grille. La structure retournée par cette fonction distante correspond aux informations de connexion du processus distant qui sont utilisées pour établir la connexion.

5.4.2 Architecture

L'extension de l'environnement d'exécution assure la connectivité entre les grappes. Il est mis en place sous forme d'un ensemble de services de grilles formant une infrastructure distribuée à plusieurs niveaux. L'architecture de cette infrastructure est représentée figure 5.6, sur deux grappes représentées chacune par un nuage.

Les services d'interconnexion sont sollicités de deux façons. Lors de l'initialisation de l'environnement et de la bibliothèque MPI, les pilotes de communication ouvrent chacun une socket sur laquelle ils écouteront les connexions entrantes. Ils déclarent alors leur point de contact en transmettant leurs informations de connexion à l'infrastructure d'interconnexion, qui les stocke. Durant l'exécution de l'application, lorsqu'un processus cherche à établir une connexion avec un autre processus, il demande ses informations de connexion à l'infrastructure d'interconnexion qui peut alors, éventuellement déclencher l'utilisation d'une technique de connectivité, ou renvoyer simplement les informations de connexion du processus distant.

5.4.2.1 Le broker

Il centralise l'information. C'est lui qui, au lancement, connaît la compositions des grappes et comment les processus s'exécutant dans chacune des grappes peuvent être contactés (quelle technique d'interconnexion est disponible). Les techniques d'interconnexion disponibles ont été décrites dans la section 5.4.3. Il doit être accessible, c'est-à-dire accepter les connexions entrantes.

5.4.2.2 La frontale

Sur chaque grappe s'exécute une frontale, qui est connectée avec les machines de la grappe (ce qui est possible car elle est située dans la grappe elle-même) et avec le broker. Les connexions sont établies dans le sens de la frontale vers le broker et des machines vers la frontale.

C'est elle qui reçoit les demandes d'informations en provenance des machines de sa grappe et éventuellement, les transmet au broker. Afin de limiter la charge sur le point central que représente le broker, la frontale dispose d'une mémoire lui permettant de conserver les informations de connexion qu'elle a déjà transmises afin de répondre à des demandes ultérieures similaires sans avoir à solliciter le broker.

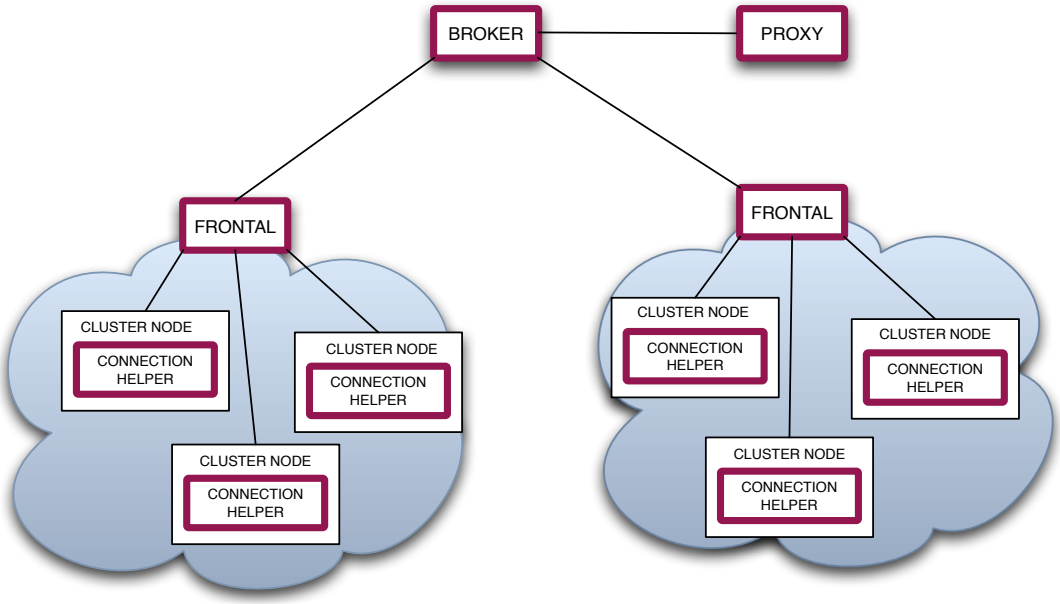


FIGURE *5.6 – Architecture de l'infrastructure des services de communications inter-grappes*

5.4.2.3 Le connection helper

Il est exécuté sur chaque machine faisant partie de la grille et permet d'utiliser des techniques d'interconnexion avancées en intervenant directement sur la machine où s'exécutent les processus de l'application. Il est connecté à la frontale de sa grappe. Les processus de l'application se connectent à lui lorsqu'ils ont besoin d'établir une communication. Un seul connection helper est présent sur chaque machine, quelque soit le nombre de processus des applications s'exécutant sur cette machine.

5.4.2.4 Le proxy

Lorsqu'il n'est pas possible d'établir une connexion directe entre deux processus, un intermédiaire est utilisé : le proxy. Les deux processus se connectent vers lui (connexion sortante), il transmet alors les messages entre ces deux processus.

Comme le broker, il doit être exécuté sur une machine accessible depuis toutes les machines de la grille.

5.4.3 Techniques d'interconnexion

Plusieurs techniques d'interconnexion sont disponibles ou ont été étudiées afin de s'adapter au plus grand nombre de situations possibles et de permettre d'établir une connexion directe entre les deux processus dans le plus grand nombre de cas possibles.

5.4.3.1 Connexion directe

On cherche en priorité à établir une connexion directe entre les deux processus, afin de n'avoir à payer aucun surcoût d'établissement de connexion. Le protocole d'établissement de connexion entre deux processus est représenté figure 5.7.

Le nœud A a besoin d'établir une connexion avec le nœud B. Il demande alors à l'environnement d'exécution comment contacter B, c'est-à-dire, quelles sont ses informations de connexion. Il se connecte donc au connection helper, qui demande à la frontale. Si la frontale ne connaît pas déjà les informations de connexion de B, elle les demande au broker.

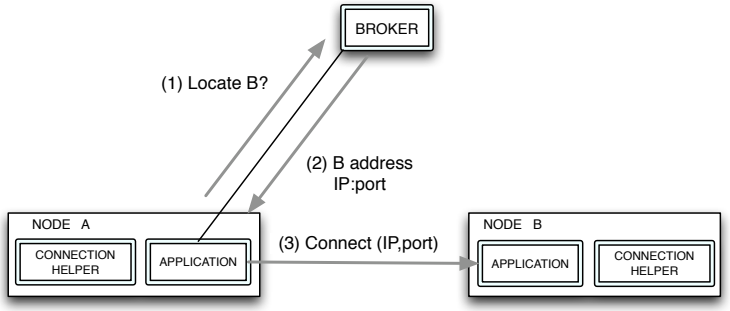


FIGURE *5.7 – Établissement d'une connexion directe entre deux processus.*

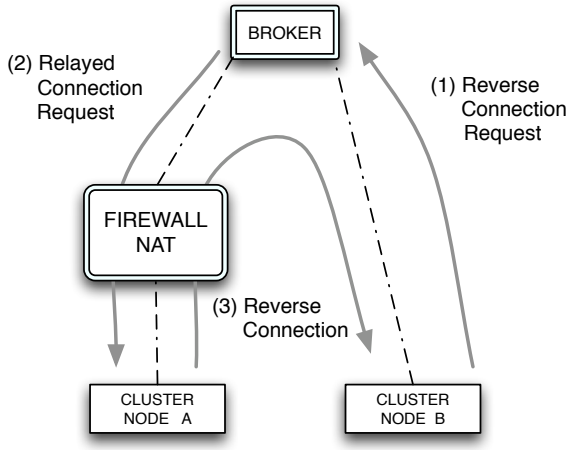


FIGURE *5.8 – Établissement d'une connexion inversée.*

Le broker lui répond en envoyant les informations de connexion de B. Elle les transmet au connection helper tout en les conservant dans sa mémoire. Ainsi, la prochaine fois que les informations de connexion de B lui seront demandées, elle pourra les envoyer sans avoir à solliciter le broker. Le connection helper reçoit les informations de connexion et les transmet au processus de l'application, qui peut ainsi établir la connexion en se connectant directement à B grâce aux informations de connexion qui lui ont été transmises.

Cette méthode est utilisable s'il existe au moins un intervalle de ports ouverts dans le pare-feu de B, ou aucun pare-feu. Il est possible de définir un intervalle de ports à utiliser par l'application et l'environnement d'exécution, correspondant à l'intervalle ouvert dans le pare-feu.

5.4.3.2 Connexion inversée

Lorsque le nœud A a besoin d'établir une connexion vers le nœud B, mais que B est protégé par un pare-feu et/ou un NAT tandis que A est accessible directement, il est possible d'inverser la connexion. Dans ce cas, ce n'est plus A qui se connecte vers B mais B qui se connecte vers A. L'inversion a uniquement lieu au moment de l'établissement de la connexion ; une fois la connexion établie, tout se déroule normalement. Le fonctionnement de ce protocole est schématisé figure 5.8.

Il s'agit ici encore d'une technique de connexion directe, n'introduisant pas de latence supplémentaire ni de diminution de la bande passante, que ce soit à l'établissement de la connexion ou durant les communications ultérieures.

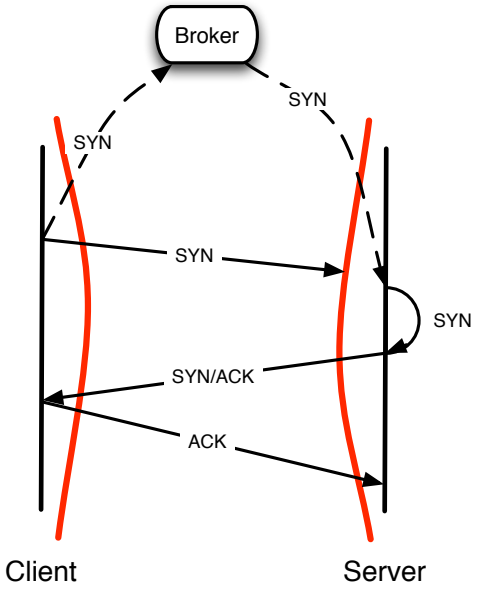


FIGURE *5.9 – Protocole Traversing TCP.*

5.4.3.3 Traversing TCP

Le protocole Traversing TCP (TTCP) est présenté dans [146] et son fonctionnement est schématisé figure 5.9. Il est utilisé pour traverser les pare-feux.

Lorsqu'un nœud A tente de se connecter à un nœud B, il envoie un paquet SYN qui est arrêté par le pare-feu de B. Le connection helper de A intercepte ce paquet SYN et le transmet au broker, qui le réinjecte dans la pile TCP de B. Le nœud croit alors avoir reçu un paquet SYN de A et lui répond par un paquet SYN/ACK. Le paquet SYN/ACK passe le pare-feu et le nœud A envoie à B son paquet ACK, qui termine l'établissement de la connexion.

Cette technique établit une connexion directe, n'introduisant pas de latence supplémentaire ni de diminution de la bande passante disponible lors des communications normales. Le seul surcoût qu'elle induit se manifeste lors de l'établissement de la communications.

5.4.3.4 TCP hole punching

Le protocole de TCP hole punching est décrit dans [78]. Il est principalement utilisé pour passer les NATs. Le nœud A et le nœud B sont tous les deux connectés avec un composant externe (le broker). Le couple *{adresse IP, port}* visible depuis l'extérieur après traduction d'adresse et le couple *{adresse IP, port}* privé est observé par le broker dans les paquets IP venant de A et de B. Ces informations sont fournies au nœud qui se connecte à l'autre afin qu'il puisse établir la connexion en passant par la règle de traduction créée à la volée pour la connexion sortante vers le broker. Le protocole de TCP hole punching est schématisé figure 5.10.

Le comportement des NATs n'étant pas normalisé, cette technique ne fonctionne pas sur toutes les configurations. La traduction d'adresse doit être cohérente et ne créer qu'une seule règle de traduction d'adresse par port ouvert dans le réseau privé. Il doit également traiter les demandes de connexion non sollicitées en ne transmettant pas le paquet SYN de manière silencieuse, sans renvoyer de paquet RST à l'expéditeur, sans quoi les demandes de connexion simultanées échoueraient.

Cette technique permet une connexion directe entre les deux processus. Elle implique un temps d'établissement de la connexion plus long qu'un établissement de connexion TCP, mais une fois la connexion établie elle ne souffre pas d'une diminution de la bande passante disponible ni d'une augmentation de la latence.

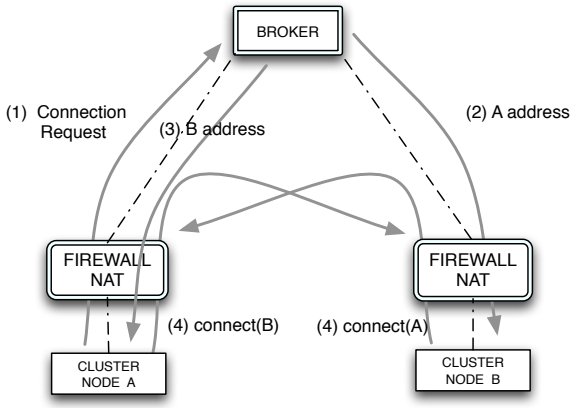


FIGURE *5.10 – TCP hole punching*

5.4.3.5 TCP splicing

Ce protocole est présenté dans [69] et sert à passer les pare-feux. Il consiste à envoyer des tentatives de connexion répétées entre les deux nœuds. Lors de la première phase, chaque nœud envoie un paquet SYN, qui est arrêté par le pare-feu. Lors de la deuxième phase, chaque nœud envoie un paquet SYN/ACK dont le numéro de séquence correspond à celui du paquet ACK envoyé par l'autre nœud augmenté de 1. Le paquet SYN/ACK passe le pare-feu et correspond au SYN/ACK qui aurait dû être répondu si le SYN avait passé le pare-feu. Le paquet ACK est alors envoyé et la connexion est établie.

Il nécessite de pouvoir réinjecter les paquets TCP SYN dans la pile TCP du processus acceptant la connexion. Le connection helper doit donc construire un paquet TCP. Or, les paquets TCP contiennent un numéro de séquence. Le connection helper n'ayant pas reçu le paquet SYN (refusé par le NAT), il ne sait pas quel numéro de séquence doit porter ce paquet. Jusqu'à il y a quelques années, les numéros de séquence des paquets TCP étaient prévisibles, ce qui rendait possible la construction et l'injection des paquets mais constituait une faille de sécurité du protocole. Ce bug a été corrigé dans la RFC 1948 [151] qui a depuis été implémentée dans toutes les piles TCP, et il n'est désormais plus possible d'utiliser cette technique.

5.4.3.6 Connexion par l'intermédiaire d'un proxy

Si aucune des techniques de connectivité disponibles n'est utilisable, les nœuds peuvent communiquer par l'intermédiaire d'un proxy. Les deux nœuds A et B se connectent au proxy qui reçoit également une commande du broker lui indiquant que ces deux nœuds doivent communiquer ensemble. L'établissement de la communication est représenté figure 5.11.

Concrètement, le broker renvoie à A les informations de connexion du proxy au lieu de lui renvoyer celles de B. Le nœud A se connecte donc au proxy et communique avec lui selon le même protocole que s'il se connectait directement à un autre nœud. Dans le même temps, B reçoit par son connection helper la commande lui indiquant de se connecter au proxy.

Dans le cas d'une connexion par un proxy, la frontale ne garde pas en mémoire les informations de connexion et fait toujours appel au broker, celui-ci devant intervenir auprès du proxy lors de l'établissement de la connexion entre les deux nœuds et le proxy.

Le proxy sert alors de relais pour les messages envoyés entre A et B. Cette technique est utilisée en dernier recours, lorsqu'il n'est pas possible d'établir une connexion directe entre les deux nœuds, car elle offre des performances de communication diminuées. Les nœuds utilisant un même proxy doivent en effet se partager la bande passante de ce proxy, et la latence est augmentée du fait du saut supplémentaire entre deux nœuds.

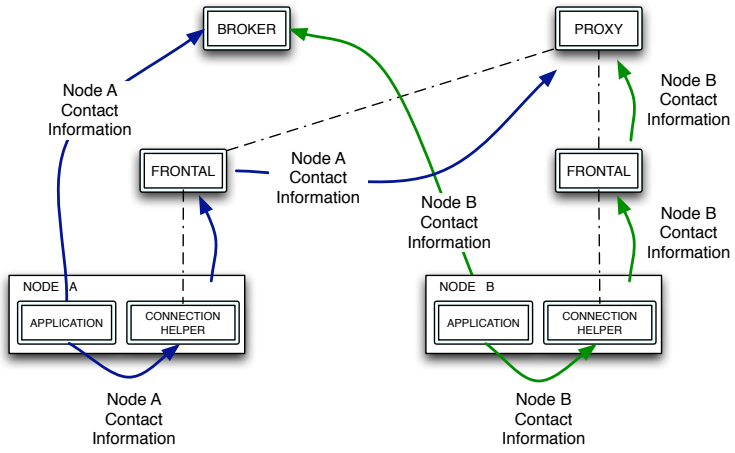


FIGURE *5.11 – Établissement d'une connexion via un proxy.*

5.5 Performances

Les performances de QCG-OMPI ont été évaluées sur Grid'5000 [41] en utilisant trois grappes :

- Orsay : grappe GdX composée de 216 machines équipées de deux processeurs Opteron 246 (2.0 MHz) et 126 machines équipées de deux processeurs Opteron 250 (2.4 MHz), reliées par un réseau GigaEthernet en utilisant 2 ou 3 interfaces réseau par machine. Certaines machines sont reliées à un réseau Myrinet 10G mais ce réseau n'a pas été utilisé.
- Rennes : grappe Paravent composée de 99 machines équipées de deux processeurs Opteron 246 (2.0 MHz) reliées par un réseau GigaEthernet en utilisant une interface réseau par machine. Les machines ont également accès à un réseau InfiniBand 10G, mais ce réseau n'a pas été utilisé.
- Bordeaux : grappe Bordereau composée de 93 machines équipées de deux processeurs Opteron 2218 (2.6 MHz) reliées par un réseau GigaEthernet en utilisant deux interface réseau par machine.

Les grappes de Orsay et Rennes sont interconnectées par une fibre noire à 10 Gb/s, celle de Bordeaux est connectée au reste de la grille par une fibre noire à 1Gb/s. La plate-forme Grid'5000 est présentée plus en détails dans la section 2.2 du chapitre 2.

Nous avons évalué les caractéristiques de QCG-OMPI en utilisant des micro-mesures spécifiques, puis nous avons évalué ses performances sur des applications standards.

5.5.1 Performances des communications

Nous avons évalué les performances des communications de QCG-OMPI en utilisant une application qui effectue un ping-pong entre les noeuds deux-à-deux. Nous avons utilisé les trois grappes citées ci-avant, en utilisant deux machines par grappe. Les deux machines situées dans la même grappe permettent de mesurer les performances des communications intra-grappe. Des communications sont effectuées entre machines de différentes grappes pour tester les communications sur la grille. Pour les expériences utilisant un proxy, celui-ci est situé dans la grappe d'Orsay. Nous avons comparé ici les performances de MPICH 1.2.7, MPICH-G2 utilisant le Globus Toolkit 4.0.5, OpenMPI 1.3a1 et QCG-OMPI.

Nous avons comparé la latence et la bande passante inter- et intra-grappe obtenues avec les différentes techniques d'interconnexion proposées par QCG-OMPI. Les résultats de latence sont résumés dans le tableau 5.2, les résultats de bande passante sont résumés dans le tableau 5.1.

On constate que OpenMPI et QCG-OMPI utilisant une connexion directe ou le protocole de Traversing TCP présentent les mêmes performances tant en latence qu'en bande passante. Une fois la connexion établie, le pilote QCG se comporte de la même façon que le pilote TCP d'OpenMPI et les performances sont les mêmes.

Le protocole de relais avec proxy de QCG-OMPI implique d'effectuer un saut de plus. Dans le cas de communications dans la grappe d'Orsay, le proxy est situé dans la même grappe que les processus, donc le

bande passante en Mb/s	QCG-OMPI direct	QCG-OMPI proxy	QCG-OMPI traversing
Intra-Grappe	894.39	894.37	894.36
Orsay-Rennes	118.48		
Orsay-Bordeaux	136.08	76.40	138.18
Bordeaux-Rennes	131.68		

TABLE *5.1 – Comparaison des bandes passantes obtenues avec les différentes techniques de connectivité de QCG-OMPI (en Mb/s)*

temps en secondes	MPICH-G2	OpenMPI	QCG-OMPI direct	QCG-OMPI proxy	QCG-OMPI traversing
Intra-Grappe	0.0002	0.0001	0.0001	0.0001	0.0001
Orsay-Rennes	0.0104	0.0103	0.0103	0.0106	0.0103
Orsay-Bordeaux	0.0079	0.0078	0.0078	0.0084	0.0078
Bordeaux-Rennes	0.0081	0.0080	0.0080	0.0287	0.0080

TABLE *5.2 – Comparaison des latences obtenues avec les différentes techniques de connectivité de QCG-OMPI (en secondes)*

surcoût en latence est très petit. Dans le cas où un des processus se trouve à Orsay (Orsay-Bordeaux ou Orsay-Rennes), le saut supplémentaire est local pour un des deux processus (celui situé à Orsay) et le surcoût en latence est faible par rapport au coût de la communication entre les deux grappes. Lorsque les processus sont situés à Bordeaux et à Rennes, les messages doivent transiter par Orsay : ce détour se ressent fortement sur la latence, qui passe de 8 ms à 28.7 ms.

Les deux processus communiquant en passant par le proxy doivent se partager la bande passante de celui-ci : on voit sur les mesures de bande passante que la bande passante obtenue en utilisant le proxy est presque divisée par 2 par rapport à la bande passante obtenue par les techniques établissant une connexion directe.

5.5.2 Scalabilité de l'environnement d'exécution

Le lancement d'une application implique des communications pour mettre en relation les processus de l'application entre eux. Concrètement, il s'agit d'échanger les points de contact des processus. C'est une phase de communications hors-bande intensives selon une certaine topologie.

La courbe 5.12 présente les temps de lancement d'une applications MPI triviale sur les trois grappes utilisées ici. Dans la mesure du possible, une proportion de deux nœuds à Orsay pour un nœud à Bordeaux et un nœud à Rennes a été respectée, jusqu'à ce que tous les nœuds disponibles à Bordeaux soient utilisés, à partir de quoi seul le nombre de nœuds à Orsay a été augmenté.

L'application utilisée est une application MPI qui s'initialise (`MPI_Init()`) et se termine (`MPI_Finalize()`). Elle implique le lancement et l'initialisation de l'environnement d'exécution, le lancement de l'application et l'échange des identifiants de connexion des processus MPI. Le `MPI_Finalize()` est implémenté comme une barrière dans l'environnement d'exécution, afin de s'assurer que tous les processus sont bien dans le `MPI_Finalize()` avant de les terminer.

Nous avons comparé ici les performances de MPICH 1.2.7, MPICH-G2 utilisant le Globus Toolkit 4.0.5, OpenMPI 1.3a1 et QCG-OMPI direct.

On peut voir que le lancement de MPICH passe mal à l'échelle. Relativement ancien, il a été conçu à une époque où les grappes étaient beaucoup plus petites (quelques dizaines de nœuds). Les gros systèmes composés de plusieurs milliers de nœuds utilisant MPICH utilisent un système de lancement dédié, plus intégré à la machine et passant mieux à l'échelle.

MPICH-G2 est basé sur MPICH et doit, en plus, utiliser les services de Globus. Les processus sont lancés

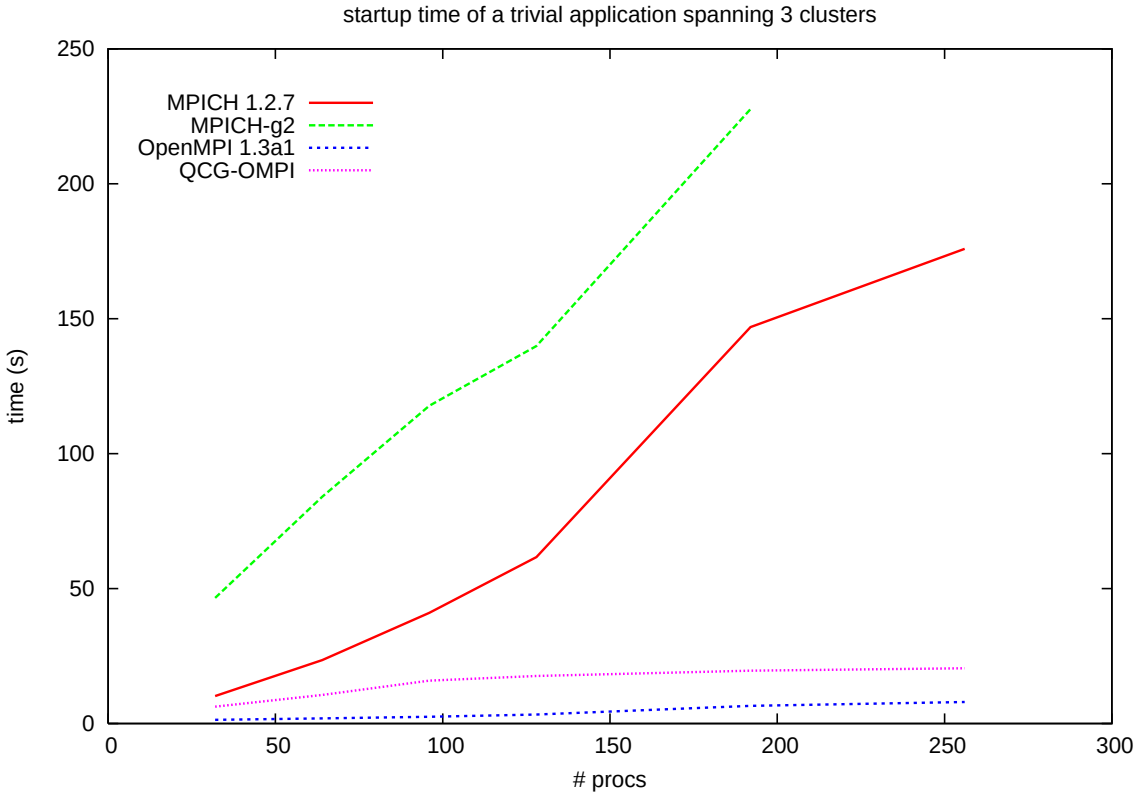


FIGURE *5.12 – Temps de démarrage d'une application triviale sur trois grappes.*

sur les nœuds par le Grid Resource Allocation and Management (GRAM), la sécurité est assurée par la Grid Security Infrastructure (GSI), l'exécutable peut éventuellement être transféré par le composant de Reliable File Transfert (RFT). Les ressources sont coordonnées par les Dynamically-Updated Request Online Coallocator (DUROC) locaux. Enfin, la bibliothèque de communication pour réseaux TCP *globus2* sert d'environnement de communication.

Les mesures de performances avec MPICH-G2 n'ont été effectuées que jusqu'à 192 noeuds pour des raisons de limitations de mémoire au niveau du noyau Linux. En effet, l'infrastructure de GT4 utilise un système de gestion de bases de données dont le serveur est situé sur le nœud maître de la grille et auquel tous les autres nœuds font appel en tant que clients. Le SGBD utilisé ici est PostgreSQL. Le serveur utilise un segment de mémoire partagée fourni par le noyau et dont la taille croît avec le nombre de clients. Au-delà de 192 clients, le serveur a besoin d'un segment de mémoire partagée dont la taille dépasse ce qu'on peut considérer comme "raisonnable".

Plus récent, OpenMPI a été conçu avec un objectif de passage à l'échelle et son temps de lancement augmente moins rapidement que celui de MPICH (logarithmiquement). On voit que QCG-OMPI présente un surcoût par rapport à OpenMPI, qui correspond à l'invocation du broker pour établir les connections hors-bande, mais son temps de lancement augmente lui-aussi logarithmiquement et est largement inférieur à celui de MPICH-G2 et même MPICH.

5.5.3 Performances sur des applications

Nous avons évalué les performances de QCG-OMPI et comparé les différentes techniques de connectivité en utilisant la suite de test NAS [16], qui fournit des applications de test spécialement conçus pour tester les environnements de calcul parallèle utilisant MPI et effectuent des opérations de calcul numérique typiques sur des matrices de différentes tailles selon la classe demandée.

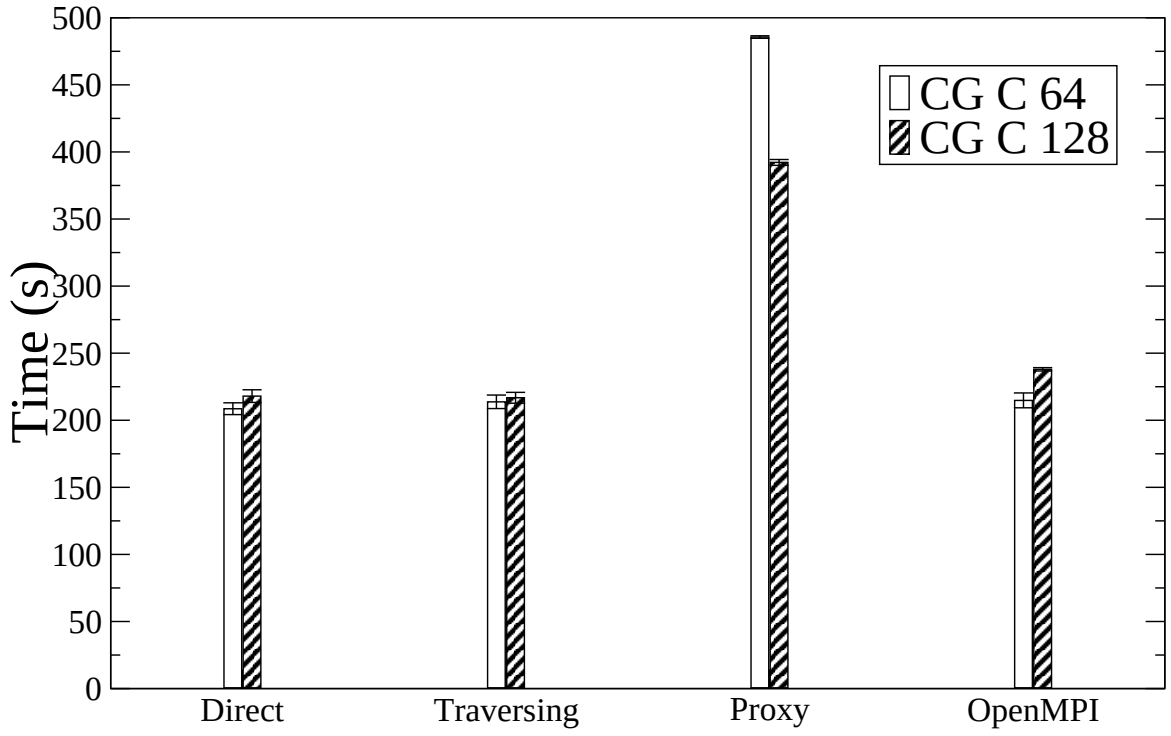


FIGURE *5.13 – NAS CG*

Il n'existe pas à ce jour d'application d'évaluation de performances équivalente aux NAS pour la grille en MPI. Les versions 3.x de la suite NAS incluent des applications pour les grilles, mais elles sont basées sur un flux de données et n'utilisent pas MPI.

Bien qu'inadaptées à un calcul efficace sur grille, ces applications s'avèrent être un bon test pour l'environnement d'exécution et la bibliothèque de communication du fait de leurs schémas de communications et de calcul inadaptés à la topologie de la grille et sollicitant d'autant plus l'intergiciel de communication.

Nous avons choisi deux applications en particulier : tout d'abord CG, qui effectue un calcul de gradiant conjugué et communique beaucoup au moyen de messages de relativement petite taille (environ deux tiers de messages de huit octets et un tiers de messages de quelques dizaines de kilo octets) et mettant donc en évidence la latence, ensuite BT, qui résout un système triangulaire par blocs et communique moins souvent que CG mais au moyens de relativement gros messages (plusieurs dizaines de kilo octets) et met donc en évidence la bande passante disponible.

Nous avons utilisé trois grappes de Grid'5000 : GdX, Bordereau et Paravent. Dans la mesure du possible, une proportion de deux nœuds à Orsay pour un nœud à Bordeaux et un nœud à Rennes a été respectée, jusqu'à ce que tous les nœuds disponibles à Bordeaux soient utilisés, à partir de quoi seul le nombre de nœuds à Orsay a été augmenté. Pour les mesures où un proxy a été utilisé, celui-ci se trouvait sur GdX.

5.5.3.1 NAS CG

La figure 5.13 compare les performances des différentes techniques de connectivité de QCG-OMPI et OpenMPI sur une exécution de CG pour la plus grande taille de matrice disponible (classe C) et sur 64 et 128 nœuds de calcul.

On constate comme attendu que les performances sont équivalentes pour QCG-OMPI direct, QCG-OMPI traversant et OpenMPI. Ces deux techniques d'interconnexion pour QCG-OMPI établissent des connexions directes entre les processus. Une fois la communication établie, les performances des communications sont équivalentes à celles d'OpenMPI, comme vu dans la section 5.5. Les applications MAS effectuant une étape d'établissement des communications (dite *"d'échauffement"*) qui n'est pas prise en compte dans la mesure, on

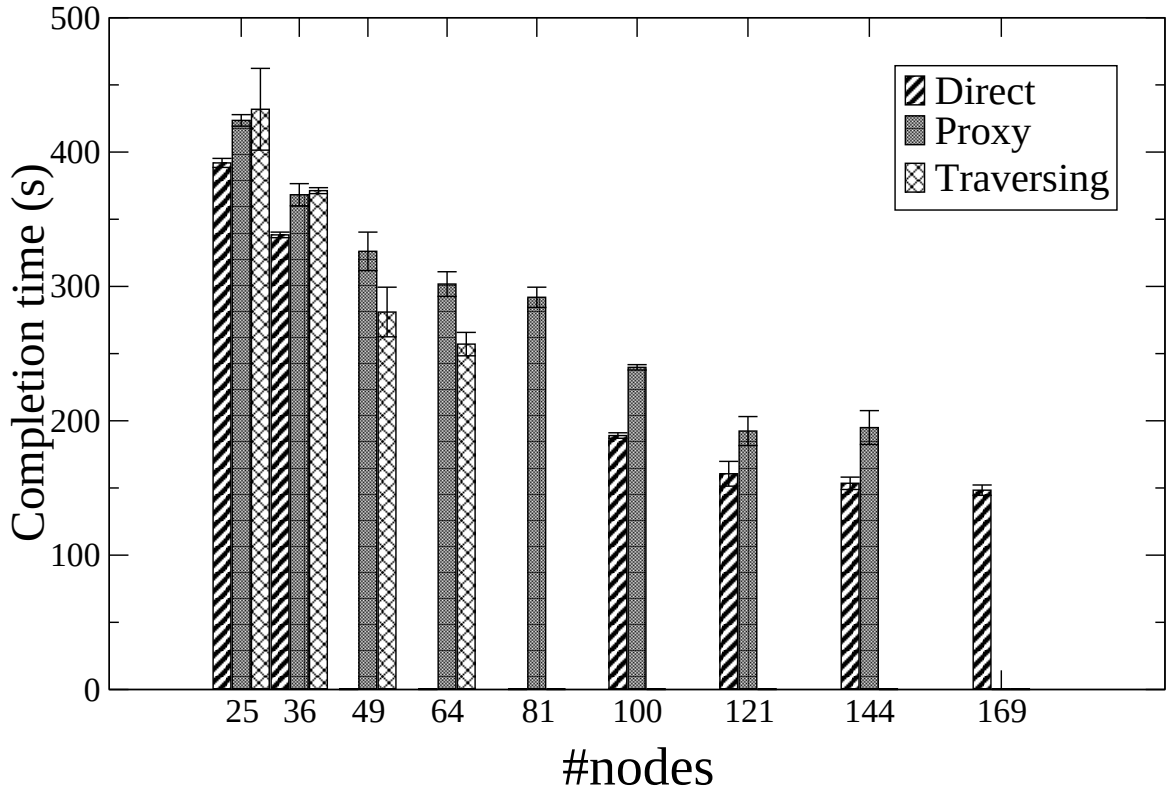


FIGURE *5.14 – NAS BT*

ne voit pas ce surcoût apparaître ici.

La méthode avec proxy présente un ralentissement d'environ 100% dû au fait que les communications entre les grappes doivent toutes transiter par un nœud qui les transfère, impliquant alors un saut de plus et augmentant la latence.

Les temps d'exécution sont très proches en utilisant 64 et 128 processus pour OpenMPI, QCG-OMPI direct et QCG-OMPI traversant. La raison de cela est que cette application ne passe pas à l'échelle sur la grille. Plus de processus engendrent plus de communications entre les grappes et ralentissent par conséquent l'application. On observe une accélération entre 64 et 128 processeurs avec QCG-OMPI proxy due au fait que les communications entre les grappes transitant par le proxy, plus de processus par grappe induisent moins de communications entre les grappes relativement au nombre total de communications et donc une sollicitation moins importante du proxy.

5.5.3.2 NAS BT

La figure 5.14 présente une comparaison du passage à l'échelle des différentes techniques d'interconnexion de QCG-OMPI sur l'application BT pour la plus grande taille de matrice disponible (classe C).

Ici encore il s'agit d'une application qui ne passe pas bien à l'échelle sur la grille. Ses performances sont limitées par les communications, ce qui nous permet ici de solliciter de manière importante la bibliothèque de communications.

On peut voir qu'à petite échelle les trois techniques (direct, proxy, traversant) sont à peu près équivalentes. Cela est dû au fait que pour un petit nombre de processus, les communications sont moins fréquentes.

La technique utilisant le traversement apparaît légèrement plus lente que les autres techniques pour 25 processus, cependant avec un écart-type important.

Quand le nombre de processus augmente, on observe qu'un écart se forme entre les performances des techniques établissant des connexions directes entre les processus et la technique relayant les messages par un proxy.

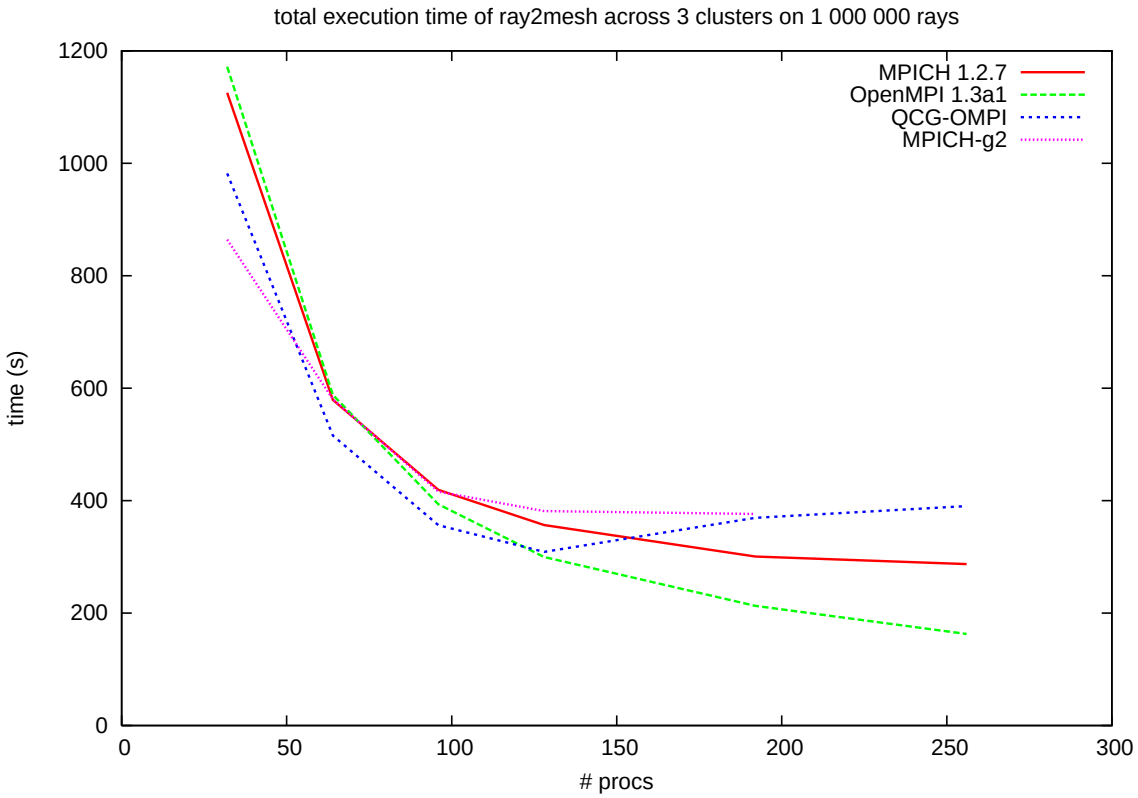


FIGURE *5.15 – Ray2mesh*

Il y a alors un plus grand nombre de messages échangés entre les grappes, sollicitant plus le proxy. Cette application échangeant des messages de taille relativement importante, ses performances sont limitées par la bande passante, en particulier en utilisant un proxy dont les processus doivent se partager la bande passante. On constate cependant un ralentissement moins important que pour l'application CG qui subit le surcoût du saut supplémentaire induit par le passage par le proxy.

5.5.3.3 Ray2mesh

Nous avons également comparé les performances de QCG-OMPI direct avec celles de MPICH 1.2.7, MPICH-G2 et OpenMPI 1.3a1 sur Ray2mesh [100], une application de géophysique comportant des phases de communications collectives et un calcul selon un schéma maitre-esclave.

La figure 5.15 présente la comparaison de leurs performances sur trois grappes de Grid'5000 (GdX, Paravent et Bordereau). Les mesures pour MPICH-G2 s'arrêtent à 192 nœuds pour les raisons invoquées dans le paragraphe 5.5.2.

Les temps mesuré ici est le temps total d'exécution de l'application entre le moment où la commande est exécutée et celui où elle retourne. Ce temps inclut le déploiement de l'environnement d'exécution, de lancement de l'application, l'établissement des communications, l'exécution de l'application elle-même et la finalisation.

On constate que MPICH et MPICH-G2 sont les plus lents et que l'écart augmente avec la taille du système. L'augmentation de cet écart est principalement dû au fait que ces deux bibliothèques ont un lancement passant mal à l'échelle. De plus, elles établissent toutes les connexions entre les processus lors de l'initialisation, ce que ne font pas OpenMPI et QCG-OMPI. Ray2mesh a un schéma de communications très régulier et nécessitant peu d'établissements de communications : l'approche de MPICH et MPICH-G2 établit donc beaucoup de communications inutiles que OpenMPI et QCG-OMPI, qui établissent à la demande les communications entre les processus, n'établissent pas.

5.6 Conclusion

Ce chapitre présente QCG-OMPI, un environnement de programmation et d'exécution d'applications MPI pouvant s'exécuter sur des grilles de calcul. QCG-OMPI s'intègre dans la pile logicielle QosCosGrid, présentée ici dans un premier temps. L'architecture de QCG-OMPI, basée sur un ensemble de services de grille fournissant des techniques de connectivité avancées, permet d'exécuter une application communiquante sur une grille de calcul de façon transparente sans nécessiter de concession sur le plan de la sécurité des ressources : des pare-feux et des mécanismes de traduction d'adresse peuvent être utilisés sans configuration particulière et sans gêner l'exécution.

Les performances de QCG-OMPI ont été évaluées par des micro-mesures et des applications réelles (bancs de test NAS, application géophysique Ray2mesh) sur plusieurs angles : scalabilité de l'environnement d'exécution (lancement), performances des communications (latence et bande passante) et temps d'exécution. La plate-forme expérimentale Grid'5000 a été utilisée comme environnement de test à grande échelle composée de plusieurs grappes. Les performances des différentes techniques de QCG-OMPI ont été comparées entre elles et par rapport à d'autres bibliothèques MPI orientées vers les grilles (MPICH-G2) ou non (MPICH, OpenMPI).

Le chapitre suivant présente comment les applications peuvent tirer parti de l'environnement d'exécution spécialisé pour les grilles de calcul, et en particulier comment l'application peut obtenir une topologie physique d'exécution optimale pour ses schémas de communications et obtenir des informations sur celle-ci auprès de l'environnement d'exécution au cours de l'exécution.

Chapitre 6

Applications MPI pour la grille

Outre les services indispensables qu'il fournit à l'application pour lui permettre de s'exécuter, l'environnement d'exécution peut également lui fournir des informations pour l'aider à s'exécuter dans le contexte particulier des grilles de calcul à grande échelle. En effet et comme vu dans le chapitre précédent, l'hétérogénéité des réseaux de communications utilisé introduit un déséquilibre dans les performances des communications de l'application. Il est donc nécessaire de prendre en compte cette topologie sous-jacente dans l'organisation des communications de l'application et, par conséquent, des calculs effectués.

6.1 Introduction

6.1.1 Définitions

On distingue deux types de topologies auxquelles sont exposées les applications parallèles : la topologie physique, due au matériel sur lequel l'application est exécutée et subie par elle, et la topologie logique, définie par l'application.

Topologie physique On définit la topologie physique comme l'organisation des ressources sur laquelle l'application s'exécute : les ressources de calcul disponibles, les performances des moyens de communications les reliant et leur répartition (graphe formé), et le positionnement géographique de ces resources les unes par rapport aux autres.

Topologie logique La topologie logique est inhérente à l'application. Il s'agit de ses schémas de communications internes, souvent conditionnés par ses schémas de calcul.

Pour obtenir des bonnes performances sur les grilles de calcul, on cherche généralement à faire correspondre la topologie logique avec la topologie physique.

6.1.2 État de l'art des systèmes de calcul sur grille

6.1.2.1 Calcul volontaire

Les applications exécutées sur les grilles de calcul communiquent généralement peu. Les plate-formes de calcul volontaire (ou bénévole) comme BOINC [5] ou XtremWeb [76] décomposent un problème en sous-tâches *indépendantes* ordonnancées sur des ressources distantes. Ces sous-tâches n'ont pas à communiquer directement ensemble, pour des raisons techniques : le calcul volontaire utilisant des ordinateurs personnels durant leurs phases d'inutilisation, il n'est pas possible de demander de synchronisation directe entre deux sous-tâches. Elles peuvent communiquer, par exemple en faisant transiter des fichiers par le serveur, et se rapprocher ainsi d'un type d'applications comme les workflows, mais les communications ne se font jamais directement entre deux clients. De plus, pour des raisons de sécurité, les connexions sont toujours établies dans le sens allant des machines de calcul (souvent situées chez des particuliers) vers un serveur central de la plate-forme de calcul, et non pas en pair-à-pair. Une synchronisation entre les tâches peut être demandée, en attendant par exemple qu'une série

de tâches ait finit d'être exécutée avant de lancer la suivante. Ici encore, la synchonisation est centralisée par le serveur. Parmi les applications BOINC, on peut citer SETI@Home, LHC@Home ou MilkyWay@Home.

Le projet de grille Européenne EGEE [117], utilisant la pile logicielle gLite [124, 47] et une infrastructure de sécurité basée sur une authentification par certificats X.509 et un service d'appartenance à une organisation virtuelle (VOMS [2]), rassemble des machines dans 70 laboratoires à travers l'Europe. Les tâches sont constituées de plusieurs processus indépendants ou ayant des dépendances connues et décrites au moyen d'un langage de description de tâches. Condor [120, 158] permet également d'exécuter des tâches découpées en sous-tâches dont les dépendances sont exprimées dans un graphe de dépendances. Une version de Condor existe spécialement pour les grilles [159] et permet de réunir des tâches Condor s'exécutant sur plusieurs grappes en une tâches de grille.

6.1.2.2 Applications communiquantes et schémas de communications

Le problème posé par l'exécution d'applications communiquantes suivant des schémas de communications quelconques (et potentiellement complexes) sur les grilles de calcul reste ouvert. Le projet QosCosGrid a pour but d'exécuter des simulations de systèmes complexes sur les grilles de calcul. Les schémas de calcul que peuvent suivre des applications parallèles ont été classés en six catégories représentées par la figure 6.1. La catégorie T0 concerne les applications qui ne communiquent pas du tout, comme les applications cibles des plate-formes de calcul volontaire. La pile gLite et Condor permettent aux applications de communiquer, cependant celles-ci doivent suivre un schéma régulier et connu à l'avance, correspondant à la catégorie T1.

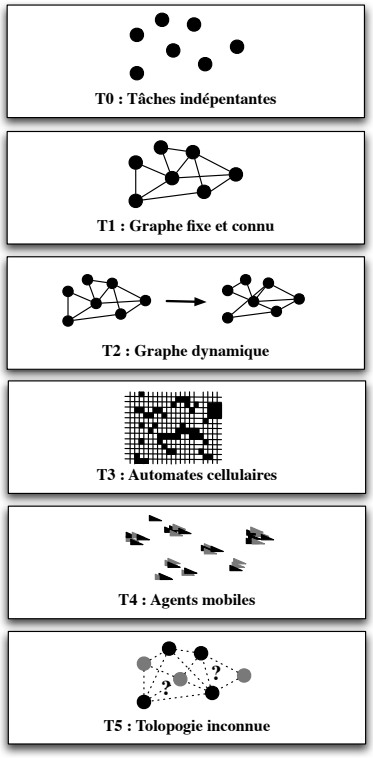


FIGURE *6.1 – Catégories de schémas de communication d'une application parallèle, des schémas les plus simples aux plus irréguliers.*

Les applications MPI peuvent suivre des schémas simples (T0, T1, T2) ou complexes (T3, T4, T5), pouvant changer au cours de l'exécution (T2, T4, T5), selon une topologie qui peut être inconnue (T5).

Le problème de l'intergiciel MPI pour les grilles a été vu au chapitre précédent ; cependant, les applications elles-mêmes doivent être adaptées au contexte des grilles de calcul pour obtenir de bonnes performances. L'application doit être adaptée aux ressources sur lesquelles elle s'exécute. La structure des grilles étant intrinsèquement hiérarchique, on s'oriente souvent vers une hiérarchisation des schémas de calcul. L'hétérogénéité des ressources nécessite également un équilibrage de charge, dynamique ou statique. L'application a donc besoin d'informations sur les ressources utilisées pour le calcul et sur leur topologie, et d'utiliser des schémas de communications et de calcul adaptés à cette topologie.

Si l'application n'est pas adaptée à la topologie physique des ressources sur lesquelles elle s'exécute, les différences de performances entre les ressources de communications et de calcul de la grille créent des déséquilibres dans l'application. Par exemple, lors d'une synchronisation les processus les plus rapides doivent attendre les plus lents, ou des communications sont ralenties par des communications plus lentes. Ce temps d'attente est du temps perdu qui pénalise les performances de l'application. On a vu dans le chapitre 5 section 5.5 avec les tableaux 5.1 et 5.2 une différence de plusieurs ordres de grandeur entre les performances des liens réseaux à l'intérieur d'une grappe et entre deux grappes, même à relativement faible distance (dans un même pays) et sur un réseau dédié (fibre noire). Une expérience de mesure de la performance obtenue par la grille EC2 [1] de Amazon montre qu'il est impossible d'obtenir une performance équivalente à celle d'une grappe sur des bancs de test standards (la suite NAS [16]) si l'application n'est pas adaptée à la topologie de l'application (inconnue dans ce cas). La mise en adéquation de la topologie logique de l'application avec la

1. Amazon Elastic Compute Cloud (Amazon EC2), `http://aws.amazon.com/ec2/`

topologie physique des ressources est alors indispensable pour limiter le ralentissement subi par les applications communiquantes.

6.1.2.3 Adaptation dynamique de l'application

Une adaptation dynamique demande à l'application de s'adapter à la topologie sur laquelle elle est exécutée. L'algorithme utilisé doit être adaptatif et utiliser des schémas de calcul et de communication suffisamment flexibles pour se plier à n'importe quelle topologie. C'est le cas de certaines implémentations de méthodes de Monte Carlo [33, 11] ou suivant le schéma maître-esclave [100]. Des techniques de pré-chargement des données pendant le calcul du paquet précédent de données peuvent être utilisées, comme dans l'amélioration du schéma maître-esclave proposée par [26]. Le schéma maître-esclave présente également un équilibrage de charge dynamique. Il s'agit ici d'application se prêtant particulièrement bien à la situation du calcul sur grilles : communiquant relativement peu, le coût de leurs communications est négligeable devant la puissance de calcul obtenue par la grille. Dans ces situations, c'est l'application qui, au moment où elle est exécutée, adapte sa topologie logique à la topologie physique.

La dynamicité et l'adaptativité sont fondamentaux pour les applications fonctionnant par vol de cycle. La bibliothèque KAAPI [90], implémentant le langage de description de tâches Athapascan [88], utilise un ordonnanceur fonctionnant sur le principe du vol de cycles pour déployer et synchroniser des tâches de calcul dont les interactions sont décrites par un graphes de dépendances. Des boucles de calcul peuvent alors être parallélisées et ordonnancées par KAAPI [67] et bénéficier du caractère adaptatif des fonctionnalités de l'ordonnanceur par vol de cycle pour être adaptées dynamiquement aux ressources disponibles. Ici encore, il s'agit d'un type d'application particulier : le calcul suit un modèle de parallélisme de données et peut être découpé en tâches Athapascan.

L'application peut obtenir des informations sur la topologie sur laquelle elle s'exécute, par exemple, pour les applications écrites en MPI, en utilisant des extensions de la norme MPI. PACX-MPI [87, 86] attribue deux rangs à chaque processus : un rang global, parmi tous les processus de l'application, et un rang local, parmi les processus s'exécutant sur la grappe locale. MPICH-G2 [111] a introduit le concept de *couleurs* : à un niveau donné, deux processus ont la même couleur si et seulement si ils appartiennent au même groupe de processus à ce niveau de hiérarchie. Il existe quatre niveaux de hiérarchie : global (tous les processus ont la même couleur), local (correspondant à une grappe, par exemple), système (correspondant à une machine) et, si disponible, le niveau de communication interne d'une machine massivement parallèle.

6.1.2.4 Adaptation statique de l'application

L'application peut aussi être adaptée statiquement, en connaissant à l'avance les ressources sur lesquelles elle va s'exécuter. La topologie physique est connue au moment où l'application est écrite et le développeur adapte la topologie logique de son application en fonction de la topologie physique. [48] propose un algorithme de tri parallèle avec décomposition initiale de l'ensemble de données. La rapidité des processeurs est connue à l'avance, une quantité de données est attribuée à chacun en fonction de leur rapidité à l'initialisation du calcul. Si la fonction de coût de l'algorithme de tri est connue, la taille des données envoyées est pondérée par la rapidité des processeurs de manière à équilibrer la charge statiquement. Si elle n'est pas connue, on construit un modèle incrémental en faisant des essais sur des jeux de données et on interpole. Le but est que tous les processeurs terminent leur phase de calcul en même temps pour être tous prêts pour la phase de communications.

La théorie de la charge divisible (Divisible Load Theory, ou DLT) [166, 165] concerne des tâches indépendantes les unes des autres, dans un modèle de parallélisme de données pur. On construit un modèle de ressources sur lequel on ordonnance les tâches. Par exemple, [165] présente la reconstruction et le calibrage de données produites par un Relativistic Heavy-Ion Collider en envoyant à chaque processeur un volume de données proportionnel à l'inverse de la fréquence du processeur. [166] établit un arbre modélisant les ressources disponibles de manière hiérarchique (suivant la hiérarchie des ressources) et qui peut se réorganiser dynamiquement selon les départs et les arrivées de ressources dans le système.

[91] s'appuie sur la routine MPI `MPI_Scatterv` pour distribuer les données en faisant l'hypothèse selon laquelle cette opération est réalisée linéairement, c'est-à-dire que les données sont envoyées à un processus une fois que les données envoyées au processus précédent l'ont été intégralement. Cette méthode propose d'ordonnancer l'envoi des données à traiter vers les processeurs effectuant les calculs en fonction des rapidités relatives des

processeurs et des bandes passantes entre le processus distribuant les données et les processus de calcul. Ceci a pour effet d'équilibrer la charge et d'optimiser le temps de calcul, avec ou sans connaissance préalable des performances des processeurs et des réseaux utilisés.

L'approche statique suivie par ces trois méthodes de découpage de l'application nécessite de connaître la rapidité des processeurs à l'avance, et donc d'effectuer un ordonnancement manuel des processus, à l'exception de [48] qui peut passer à une approche dynamique si cette information n'est pas connue.

6.1.2.5 Discussion et adaptation de la topologie physique à la topologie logique

Il est difficile d'écrire une application s'adaptant dynamiquement à toutes les topologies physiques et fournissant un équilibrage de charge dynamique. Il faut alors concevoir une application capable de s'adapter à toute topologie physique, ce qui est très difficile et parfois impossible pour certains calculs. Les approches statiques, en connaissant la topologie physique au moment où l'application est écrite, sont plus génériques en termes de type d'application.

Cependant, si l'application est prévue pour s'exécuter sur une topologie physique particulière, il est nécessaire de donner la possibilité à l'utilisateur de retrouver cette topologie physique lors des exécutions de l'application. Pour cela, il est possible de mettre à contribution le méta-ordonnanceur de grille. En effet, celui-ci a une connaissance des ressources disponibles sur la grille suffisante pour être capable d'instancier une topologie paramétrée par des conditions sur les ressources.

Par exemple, si on suppose que l'on dispose de 200 processeurs répartis sur trois grappes : une grappe constituée de 120 processeurs et deux grappes de 40 processeurs chacune. Cette situation est représentée figure 6.2. L'approche de MPICH-G2 consiste à présenter à l'application la topologie physique ; on séparera alors les processus exécutés sur cette configuration en trois groupes, un de 120 processus et deux de 40 processus chacun. Globus utilise le méta-ordonnanceur de grille GridWay [103, 15] qui permet de définir des conditions sur les ressources de calcul (fréquence du CPU, type d'architecture...) qui doivent être remplies par *tous* les nœuds attribués par le méta-ordonnanceur, mais ne permet pas de définir des groupes de processus répondant à des conditions particulières, ni de définir de conditions sur les ressources réseaux entre des processus donnés (pas de définition de conditions à grain fin). Il n'est donc pas possible de spécifier de plan de partitionnement des processus au moment d'instancier la topologie physique pour allouer des ressources à l'application.

Cependant, il est possible que l'application ne puisse pas utiliser de communicateurs de tailles différentes, ou que ce déséquilibrage physique crée un déséquilibrage de charge. On cherchera alors à utiliser cinq communicateurs de 40 processus chacun. L'approche QosCosGrid n'utilisera une telle topologie que si des communicateurs de quarante processus chacun ont été prévus par l'application. Si les deux grappes #2 et #3 sont interconnectées par un réseau répondant à des caractéristiques définies par le programmeur de l'application, par exemple si elles sont interconnectées par un réseau local suffisamment rapide, les processus s'exécutant sur ces deux grappes pourront aussi être placés dans le même communicateur. Toutes ces possibilités sont laissées à la disposition du développeur de l'application, qui dispose alors de plus de maîtrise de la topologie sur laquelle son application va être exécutée et peut donc construire ses schémas de calcul avec plus de liberté pour équilibrer lui-même la charge de son application.

Afin de ne pas nécessiter la capacité à s'adapter à n'importe quelle topologie de la part de l'application, on peut mettre à contribution l'ordonnanceur et lui demander d'attribuer une topologie compatible avec ses schémas de communication pour exécuter l'application. La participation de l'ordonnanceur est détaillée dans la section 6.2. L'ordonnancement des communications est d'abord décrit dans les communications collectives de MPI dans la section 6.3, puis dans deux exemples d'applications dans la section 6.4.

6.2 Topologie

Une application parallèle pour grilles de calcul est développée en ayant à l'esprit une topologie particulière. Les schémas de calcul et de communications qu'elle suit s'appliquent particulièrement bien à une certaine topologie et les processus peuvent être regroupés selon des caractéristiques de communications (par exemple, des processus communiquant souvent entre eux) ou de besoins en ressources (par exemple, nécessité de disposer de beaucoup de mémoire).

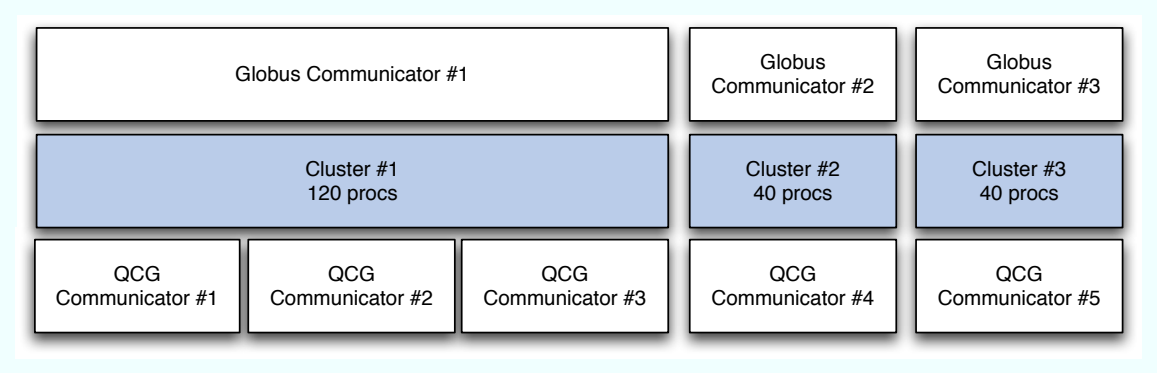


FIGURE *6.2 – Construction de communicateurs sur une topologie : l'approche MPICH-G2 et l'approche QCG.*

rang	0	1	2	3	4	5	6	7	8	9	10	11	12
profondeur	2	3	3	3	3	3	3	3	3	3	3	3	3
couleurs	W	W	W	W	W	W	W	W	W	W	W	W	W
	M	E1	E1	E1	E1	E2	E2	E2	E2	E3	E3	E3	E3
		C1	W1	W1	W1	C2	W2	W2	W2	C3	W3	W3	W3

FIGURE *6.3 – Tableau des couleurs correspondant à la topologie définie par le JobProfile donné par la listing A.1.*

6.2.1 Description de la topologie

Ces besoins sont exprimés et formalisés par le développeur dans un fichier compagnon de l'application : le JobProfile décrit la structure de l'application afin d'apporter des indications au méta-ordonnanceur qui attribuera les ressources pour l'application. Cette description s'inspire du modèle présenté dans [116]. Un exemple de JobProfile est donné par le listing A.1 de l'annexe A.

Dans cet exemple, le programmeur décrit la structure topologique et les besoins en ressources d'une application constituée de treize processus. La première partie décrit des patrons de ressources (*resourceTemplates*) nommés et définissant des conditions pour le matériel utilisé. Par exemple, on peut définir un patron fixant une latence maximale, une bande passante minimale, une fréquence de microprocesseur minimale, ou une quantité de mémoire disponible minimale. Ces patrons sont nommés pour être utilisés dans la description des ressources demandées pour la tâche.

On décrit ensuite la tâche elle-même. Les ressources sont décrites en définissant les groupes de processus qui constituent la topologie. Ici, le groupe *"monde"* contient tous les processus. Il contient les groupes *"maître"*, constitué d'un seul processus, et les groupes *"esclaves1"*, *"esclaves2"* et *"esclaves3"*. Chaque groupe *"esclavesX"* contient un *"contremaîtreX"*, relié au *"maître"* par une connexion dont la bande passante minimale est définie, et aux processus du groupe *"travailleursX"* par une connexion dont la latence maximale est définie. Tous les processus doivent disposer d'une quantité minimale de mémoire définie, et ce minimum est plus élevé pour les processus *"maître"* et *"contremaîtreX"*. Enfin, les processus des groupes *"travailleursX"* doivent disposer d'un CPU cadencé à une fréquence minimale définie. La description de la tâche définit ainsi les groupes de processus, mais également les besoins en interconnexion entre les processus, au sein d'un groupe et entre les groupes, et en ressources de calcul. Des variables du JobProfile peuvent être affectées par le méta-ordonnanceur de grille au moment où la topologie est instanciée, comme par exemple le nombre de procesus total (utilisé dans l'exemple de JobProfile A.1 : $\$NPROCS$, ou l'identifiant de la tâche $\$JOBID$).

Les groupes auxquels les processus appartiennent sont représentés sous forme d'un tableau figure 6.3. Le nom du groupe auquel un processus appartient à un niveau de hiérarchie donné est appelé sa *couleur*. Le nombre de couleurs auxquelles un processus a accès, qui correspond au nombre de groupes auxquels il appartient, est appelé sa *profondeur*. Par soucis de compacité, les groupes "esclaveX" sont notés "EX", les groupes "travailleursX" sont notés "WX", les groupes "contremaîtreX", sont notés "CX", le groupe "maître" est noté "M" et le groupe "monde" est noté "W".

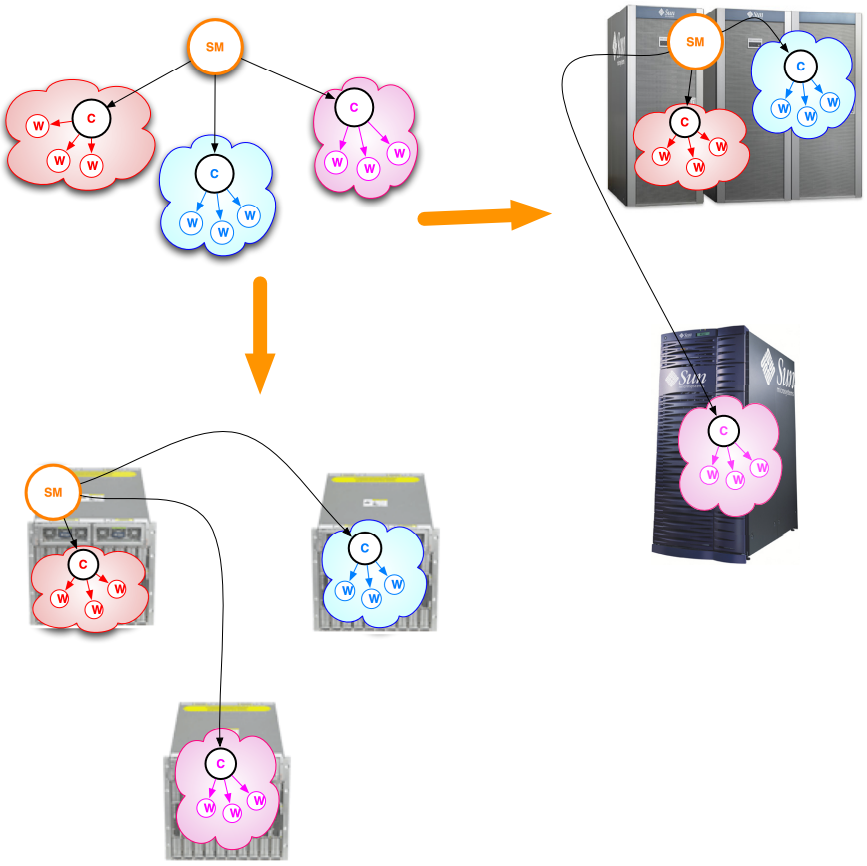


FIGURE *6.4 – Transposition d'une topologie logique (en haut à gauche) en deux topologies physiques correspondantes et correctes : trois grappes, comme demandé par la topologie logique, ou deux grappes, en regroupant les processus de deux groupes sur une seule grappe.*

Enfin, le JobProfile décrit les conditions d'exécution de l'application : chemin vers le fichier exécutable (URI permettant sa localisation dans la grille), arguments, chemin vers les fichiers de données à passer en entrée, nombre de processus, chemins vers les fichiers vers lesquels seront redirigées les sorties standards. . . .

6.2.2 Instanciation de la topologie par le méta-ordonnanceur

Le JobProfile décrit une topologie logique qui est celle de l'application. Il est passé au méta-ordonnanceur de grille qui est chargé d'attribuer des ressources matérielles compatibles avec les conditions demandées. Celui-ci doit donc instancier une topologie physique répondant *au moins* aux conditions définies par le programmeur et correspondant donc à la topologie logique demandée. Des exemples de correspondances entre topologie logique et topologie physique sont représentés figure 6.4. On voit en haut à gauche la topologie logique d'une application constituée de quatre groupes de processus : un processus seul (en orange, noté SM) et trois groupes de plusieurs processus. Le méta-ordonnanceur peut décider d'ordonnancer le processus orange et le groupe rouge sur une grappe, et chacun des deux autres groupes (bleu et rose) sur une autre grappe. C'est le cas représenté en bas à gauche de la figure. Si une grappe qui contient suffisamment de ressources pour accueillir entièrement deux groupes (par exemple, bleu et rouge) est disponible, le méta-ordonnanceur peut également décider d'ordonnancer deux groupes sur la grappe de grande taille, et le processus orange et le troisième groupe (rose) sur une grappe plus petite. Si la grappe la plus grande dispose de suffisamment de ressources, il peut aussi ordonnancer le processus orange dessus. C'est le cas représenté à droite de la figure. Le mécanisme d'ordonnancement pour la grille est décrit dans [17].

Ces informations permettent également au méta-ordonnanceur de grille d'optimiser l'organisation dans le temps d'un ensemble de tâches afin de réduire le temps total de cet ensemble de tâches (*makespan*). Il dispose alors de plus d'informations pour organiser les tâches en fonctions du partitionnement des ressources utilisables et ainsi paramétrer la combinatoire de l'ordonnancement.

Les applications exécutées plusieurs fois en changeant le nombre de groupes de processus (typiquement, les applications utilisant des algorithmes évolutionaires, en faisant varier le nombre d'ilots de populations) peuvent particulièrement bénéficier de cette meilleure combinaison des tâches. L'ordre des applications peut être modifié afin de faire passer, par exemple, une exécution utilisant huit groupes en même temps qu'une éxécution utilisant un seul groupe sur le reste des ressources (l'exécution demandant les ressources pouvant être les plus partitionnées en même temps que celle demandant des ressources le moins partitionnées, plus exigeante), puis des exécutions utilisant deux et quatre groupes (deux exécutions moyennement exigeantes en matière de partitionnement).

Le mécanisme d'adaptation de la topologie physique des ressources allouées par le méta-ordonnanceur de grille à l'application programmée selon des chémas de communications et de calculs hiérarchique a été publié dans [17].

6.2.3 Transmission des informations de topologie à l'application

Au cours de l'exécution, les processus peuvent avoir besoin de savoir à quel groupe de processus ils appartiennent à une profondeur donnée. Ces informations sont transmises à l'application par l'intergiciel de communication à travers l'environnement d'exécution.

La topologie est instanciée par le méta-ordonnanceur. Il établit alors la correspondance entre les processus et les groupes auxquels ils appartiennent. L'environnement d'exécution de l'application parallèle transmet ces informations aux processus de l'application au cours de l'exécution afin que l'application puisse les utiliser et s'y adapter. Grâce à la participation du méta-ordonnanceur, l'application dispose d'un connaissance a priori de la topologie sur laquelle elle s'exécute et ses schémas de communications et de calculs sont déjà adaptés à la topologie physique sous-jacente : l'adaptation lors de l'exécution est donc beaucoup moins complexe que si l'application devait prendre en compte une topologie quelconque.

Les noms de groupes alphanumériques sont transmis tels quels à l'application. Il est donc possible pour un processus de savoir exactement dans quel groupe il se trouve. Par exemple, si le programmeur de l'application définit deux groupes, l'un devant effectuer un calcul matriciel et appelé "matrix" et l'autre devant effectuer un traitement d'image et appelé "image", les processus peuvent déterminer dans quel groupe ils se trouvent en comparant sa couleur aux chaînes de caractères possibles. Dans l'exemple de topologie définie ci-avant, les groupes de processus nommés "travailleursX" effectuent les calculs. Les processus peuvent savoir s'ils appartiennent à ce groupe en effectuant une comparaison de chaînes de caractères.

Pour des raisons pratiques (de programmation), il peut être nécessaire de disposer de la couleur sous forme d'un entier. Ce mode de représentation étant moins explicite qu'une chaîne de caractères, c'est la chaîne de caractères telle qu'elle est définie par le programmeur qui est transmise directement. MPICH-G2 transmet la topologie à l'application sous forme d'un tableau d'entiers. Les seules opérations possibles sur ces couleurs sont donc des opérations de comparaison. Les entiers fournissent une représentation abstraite de la topologie, tandis que des chaînes de caractères donnent une connaissance concrète du groupe auquel les processus appartiennent. Par exemple, la représentation des couleurs sous forme d'entiers permet de les utiliser pour construire les communicateurs adaptés à la topologie en utilisant la routine MPI `MPI_Comm_split`.

MPICH-G2 fournit à l'application une représentation abstraite de la topologie physique sur laquelle elle s'exécute, car cette topologie est subie par l'application et non pas dictée par elle. L'approche QosCosGrid demande aux ressources physiques de s'adapter à l'application, et la transmission des couleurs aux processus de l'application leur permettent de retrouver leur place dans cette topologie logique ; l'approche MPICH-G2 propose à l'application de s'adapter à une topologie physique, et la transmission des couleurs à l'application lui permettent de connaitre cette topologie.

Il est possible d'obtenir ces couleurs sous forme d'entiers grâce à une fonction de conversion. Cette fonction effectue une bijection de l'ensemble des couleurs alphanumériques vers un intervalle de valeurs entières comprises entre 0 et N-1, si N est le nombre de couleurs différentes définies par le programmeur. Il s'agit d'une fonction de l'interface de programmation de QCG-OMPI, étendant l'interface MPI implémentée par QCG-OMPI. Son

Listing 6.1 – Prototype de la fonction de conversion d'une couleur vers un entier

```
int QCG_ColorToInt( char* color );
```

prototype est présenté par le listing 6.1.

Dans ce qui suit, on appellera *groupe* de processus un ensemble de processus appartenant au même sous-ensemble à un niveau de hiérarchie donné. Concrètement, à un niveau de hiérarchie donné, il s'agit de l'ensemble de processus ayant la même couleur.

6.2.4 Équilibrage de charge statique

Les conditions sur les ressources de calcul données dans le JobProfile et la possibilité de connaitre les tailles des communicateurs permettent au programmeur de prévoir l'équilibrage de charge de l'application au moment de sa conception.

Dans l'exemple donné par la figure 6.2, le découpage des communicateurs découverts au moment de l'exécution fait par MPICH-G2 peut créer un déséquilibrage de charge, si le calcul à effectuer sur le communicateur #1 représente la même charge que le calcul à effectuer sur les communicateurs #2 et #3. L'approche QosCosGrid permet d'équilibrer statiquement la charge de l'application en permettant de fixer la taille des communicateurs lors de la conception de l'application. Ainsi, sur la figure 6.2, les cinq communicateurs QCG ont une taille égale, sur des machines équivalentes, et les groupes de processus qui les composent peuvent effectuer un calcul équivalent.

On parle alors d'équilibrage de charge statique, réalisé au moment de la conception de l'application, par opposition à l'équilibrage de charge dynamique nécessité par une découverte de la topologie lors de l'exécution.

Les applications pour lesquelles cette approche fonctionne le mieux sont celles des catégories T0, T1 et T3, dans les catégories définies figure 6.1. On peut imaginer des techniques permettant de supporter d'autres catégories qui seront évoquées en conclusion et ouverture de ce chapitre.

6.2.5 Adaptation des schémas de communications à une topologie hiérarchique

Le principe de base de l'adaptation d'applications à la topologie physique sous-jacente est de réduire le nombre de communications coûteuses.

Par exemple, une décomposition de domaine hiérarchique peut se faire comme illustré figure 6.5. Le domaine est décomposé en trois sous-domaines. Le calcul sur chaque sous-domaine est effectué sur une grappe. Puis, au sein de chaque grappe, chaque sous-domaine est découpé en sous-domaines de plus bas niveau, et chacun va être pris en charge par un processus.

Les communications pour le calcul de chaque sous-domaine de niveau grappe vont s'effectuer au sein de la grappe où il est calculé (flèches verticales). Les communications de plus haut niveau, entre les sous-domaines de niveau grappe, se font entre des processus élus "maîtres" dans leur grappe (flèches horizontales).

Les sous-domaines calculés par les processus étant plus petits et étant des sous-ensembles du sous-domaine calculé par la grappe à laquelle ils appartiennent, les communications ont lieu plus fréquemment au sein de la grappe. Les communications entre grappes sont généralement moins fréquentes, par exemple pour échanger des régions fantômes une fois que le calcul de l'itération actuelle a été réalisé par tous les processus.

La structure hiérarchique d'une application parallèle destinée à être exécutée sur des grilles de calcul peut être représentée comme sur la figure 6.6. Le plus bas niveau de communications est le cœur. Une communication entre deux cœurs d'une même machine est la plus rapide, en terme de bande passante comme de latence. Le bus système est utilisé comme moyen de communication entre les cœurs. Cependant, il est partagé entre tous les cœurs de la machine.

Pour communiquer avec les processus d'un autre groupe (exécutés sur une autre machine, ou dans une autre grappe...), dans le cas d'un algorithme hiérarchique, les processus élisent un maître qui est chargé des communications avec les autres processus au niveau supérieur de la hiérarchie.

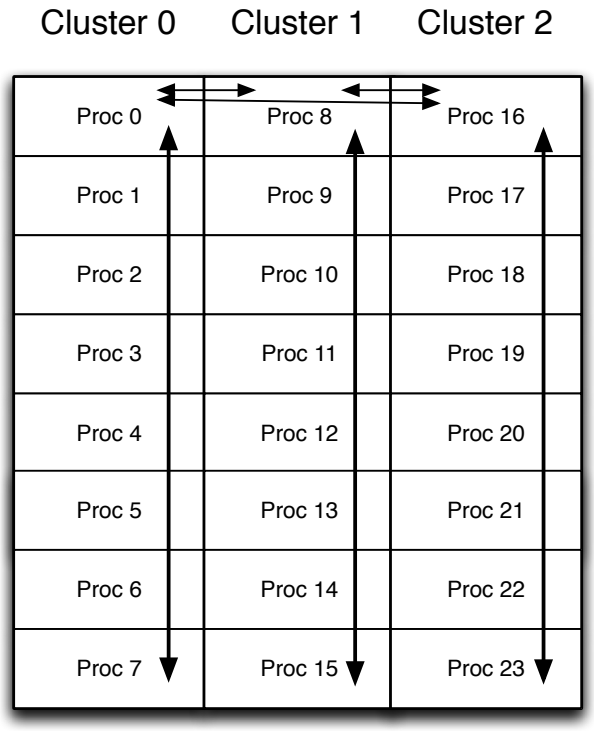


FIGURE 6.5 – *Décomposition de domaine hiérarchique pour 24 processus uniformément répartis dans trois grappes.*

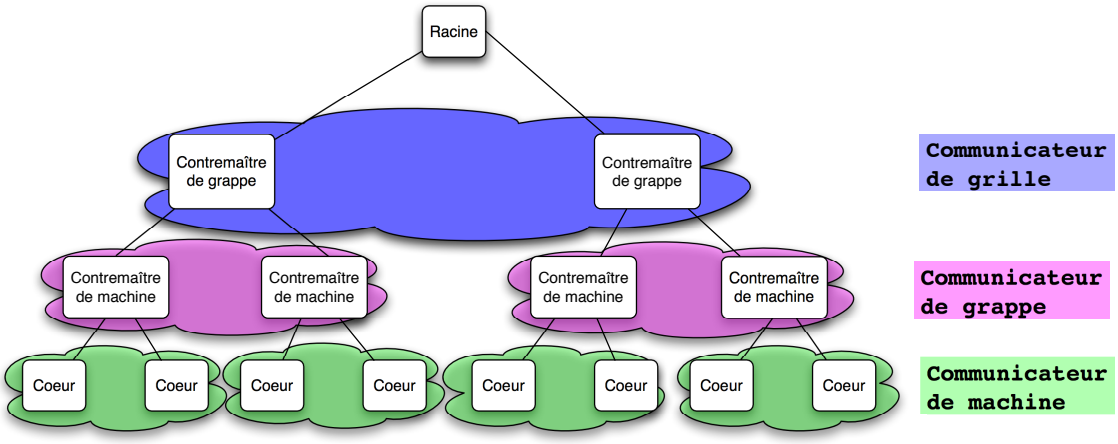


FIGURE 6.6 – *Structure hiérarchique d'une application parallèle pour les grilles de calcul.*

6.3 Communications collectives

Les communications collectives sont un type de communications important utilisé par les applications distribuées, et une part importante de la norme MPI. Leur optimisation a fait l'objet de nombreux travaux ces vingt dernières années

Une analyse de traces sur une machine Cray T3E 900-512 utilisée en production a permis d'étudier le temps passé dans les appels de fonctions MPI par les programmes exécutés sur cette machine [141]. Durant la période d'observation, 10 289 exécutions de programmes MPI ont été analysées. Il a été constaté que 8,54% du temps total d'exécution était passé dans des appels à des fonctions MPI. De plus, presque tous les programmes MPI utilisent des opérations collectives, alors que les programmes utilisant des communications point-à-points ne représentent que 18,9% du temps de calcul de la machine. Les opérations collectives représentent 34 251 des 58 260 heures passées dans des fonctions MPI, soit 58,9%.

Les communications collectives utilisent des schémas de communications qui, comme pour les applications MPI en général, peuvent être adaptés aux particularités des grilles de calcul, en particulier à leur topologie hiérarchique.

Nous verrons dans un premier temps des algorithmes utilisés couramment pour les opérations collectives, dans la section 6.3.1, puis comment ces opérations peuvent être hiérarchisées tout en utilisant ces algorithmes bien connus à chaque niveau de hiérarchie dans la section 6.3.2. Les performances des opérations hiérarchisées seront évaluées et comparées aux opérations non hiérarchisées dans la section 6.3.3.

6.3.1 Algorithmes de communications

Les algorithmes présentés ici concernent les principales opérations collectives définies par la norme MPI et sont ceux qui sont actuellement les plus utilisés dans les implémentations actuelles de la norme MPI et utilisés dans la suite. Une revue plus exhaustive des algorithmes disponibles peut être trouvée dans la thèse de Jelena Pješivac-Grbović [134].

6.3.1.1 Diffusion (MPI_Bcast)

L'importance de ces fonctionnalités de communications a motivé des efforts de recherche visant à leur optimisation. Une diffusion consiste à faire parvenir des données à chaque processus participant à la diffusion. Le processsus à l'origine de l'opération, qui est le seul à disposer de la donnée au moment du démarrage de la diffusion, est dit processus *racine*. Une diffusion est terminée une fois que tous les processus autres que la racine ont reçu cette donnée. On peut citer les algorithmes de diffusion en arbres de Fibonacci (arbres k-aires) et binomiaux pour les diffusions et les arbres binaires-découpés [160], qui sont une optimisation en bande passante des arbres binaires : la racine de la diffusion sépare le message en deux et envoie une moitié à chaque côté de l'arbre. Chaque moitié de l'arbre diffuse la moitié du message, et chaque nœud échange sa moitié avec un nœud de l'autre moitié de l'arbre.

6.3.1.2 Réduction (MPI_Reduce)

Les opérations de réduction sont elles aussi très importantes, puisqu'elles représentent 40,7% du temps passé dans des fonctions MPI (37,0% pour la fonction MPI_Allreduce et 3,7% pour la fonction MPI_Reduce). Elles consistent à effectuer une opération sur des données disponibles dans tous les processus participant à la réduction et à faire parvenir le résultat à un processus dit *racine* de la réduction ; par exemple, un calcul de maximum global, ou une somme globale. L'opération est terminées une fois que le résultat de l'opération sur les données de tous les processus participant à la réduction est disponible dans la racine. On utilise ici encore souvent un arbre de Fibonacci, en optimisant la bande passante en découpant et en doublant les messages envoyés [142].

Pour des messages de grande taille, une optimisation en bande passante consiste à utiliser un algorithme de chaîne. Chaque nœud reçoit d'un de ses voisins immédiats une portion du message. Il la traite et transmet le résultat à son autre voisin puis attend la portion suivante, jusqu'à la fin du message. Cet algorithme établit un pipeline dans la chaîne et permet d'obtenir une plus grande bande passante.

6.3.1.3 Concaténation (MPI_Gather et MPI_Allgather)

Une concatenation consiste à rassembler dans un tampon des donnés disponibles sur chacun des processus participant à la concaténation. Dans le cas où un seul processus dispose du résultat à la fin de l'opération (MPI_Gather), celui-ci est alors appelé *racine* de la concaténation. L'opération est teminée lorsqu'il dispose des données envoyées par tous les processus y participant. Dans le cas où le résultat est redistribué (MPI_Allgather), tous les processus participant à l'opération le reçoivent et aucun processus n'est considéré comme racine ; l'opération se termine lorsque tous les processus participant à l'opération disposent des données envoyées par tous les processus.

Les arbres sont aussi utilisés pour les concaténations de messages, à ceci près qu'on utilise des arbres ordonnés afin de préserver l'ordre des messages dans le tampon résultant de leur concaténation.

L'algorithme de Bruck [34] est utilisé pour les opérations de concaténation de messages assurant que le tampon obtenu est fourni à tous les participants (MPI_Allgather). À chaque étape i de l'opération, chaque nœud de rang r échange le tampon dont il dispose localement avec le nœud de rang $r + 2^i$. Il concatène le message reçu avec le contenu de son tampon local et utilise le tampon résultant pour l'étape suivante. Cet algorithme effectue l'opération en $\lceil log_2 N \rceil$ étapes si N est le nombre de participants à l'opération.

6.3.1.4 Barrière (MPI_Barrier)

Une barrière permet de synchroniser des processus. Elle assure qu'aucun des participants à la barrière n'en sort avant que tous les participants n'y soient entrés. En pratique, on peut utiliser une concaténation de messages avec diffusion des messages à tous les participants en utilisant des messages de taille nulle. On peut aussi utiliser un double anneau : les nœuds font tourner un jeton dans un anneau dans le sens horaire pour le premier tour, puis, lorsqu'ils reçoivent le jeton pour la seconde fois, le font tourner dans le sens trigonométrique et sortent de la barrière dès que le message est transmis. Cet algorithme requiert $2N$ opérations, et passe donc moins bien à l'échelle que l'algorithme de Bruck. On peut aussi utiliser un algorithme dit de "*fan-in-fan-out*" : une racine est déterminée parmi les participants ; tous les participants lui envoient un message de taille nulle en utilisant un algorithme de concaténation, et une fois tous les messages reçus la racine diffuse un message de taille nulle vers tous les participants, qui peuvent sortir de la barrière une fois ce message reçu. Suivant les algorithmes utilisés pour la concaténation et la diffusion, ces algorithmes on besoin de $log_2 N$ à N étapes pour s'exécuter.

6.3.1.5 Distribution (MPI_Scatter)

Une distribution (MPI_Scatter) consiste à envoyer des données contenues dans un tampon par le processus à l'origine de la distribution (on parle de *racine* de la distribution) vers les processus participant à l'opération. Les données envoyées à chaque processus sont spécifiques à chacun : c'est en celà qu'une distribution se différencie d'une diffusion, où tous les processus reçoivent la même donnée.

Les algorithmes de distribution utilisent des arbres binomiaux ordonnés afin de préserver l'ordre des messages et d'assurer un routage correct vers les destinataires.

6.3.1.6 Échange de messages (MPI_Alltoall)

Lors d'un échange de message entre les processus participants à l'opération (MPI_Alltoall), chaque processus envoie des messages à tous les processus. La spécification d'une opération d'échange de messages entre tous les processus participant à l'opération diffère d'une concaténation avec redistribution du résultat auprès de tous les participants du fait que les participants ont la possibilité d'envoyer un message différent à chaque participant.

L'algorithme de Bruck peut être utilisé pour des messages de petite taille. Pour des messages de taille moyenne, cette opération est parfois implémentée en postant des opérations d'envois et de réceptions non-bloquants vers les autres participants et en attendant qu'elles soient toutes effectuées. Pour des gros messages et si le nombre de participants est une puissance de deux, un algorithme d'échange par paire est utilisé. À l'étape i, chaque nœud de rang r envoie son message $r + r \oplus i$ au nœud de rang $r + r \oplus i$ ($\oplus$ symbolisant le "ou" bit-à-bit) et son message $r - r \oplus i$ au nœud de rang $r - r \oplus i$. Cet algorithme nécessite N étapes, si N est le nombre de participants. Il ne fonctionne pas si N n'est pas une puissance de 2.

6.3.2 Communications collectives hiérarchiques

Comme signalé dans [142], ces algorithmes fonctionnent de manière optimale dans des environnements homogènes, et pour la plupart, quand le nombre de participants est une puissance de deux. Leurs performances sont sévèrement dégradées par des temps de transmissions des messages non-homogènes et, dans la plupart des cas, par un nombre de participants qui n'est pas une puissance de deux.

Dans le contexte des grilles de calcul, les temps de transmissions des messages peuvent parfois différer de plusieurs ordres de grandeur suivant la localisation géographique des nœuds communiquant ensemble. Il est donc nécessaire d'adapter les algorithmes de communications à la topologie physique des grilles afin de prendre en compte cette hétérogénéité.

Le problème de la réduction en milieu hétérogène a été approché par [122] en considérant un modèle où les messages peuvent être transmis par des nœuds ayant des temps de traitement hétérogènes. Ce modèle a un sens dans le cas d'environnements hétérogènes en termes de vitesse de traitement des messages, les opérations de réduction impliquant une opération locale sur le message. Cependant, le modèle utilisé ne prend pas en compte la possibilité d'une hétérogénéité des temps de communications point-à-point.

Un modèle théorique plus détaillé est présenté dans [42]. Le modèle considéré tient compte des coûts de communications hétérogènes et des différences dans la rapidité de traitement des messages. En considérant une approche à plusieurs niveaux de hiérarchie, l'algorithme proposé minimise la complexité d'une instance du problème de diffusion pondérée en proposant un algorithme de diffusion glouton.

Comme présenté dans [110] et dans la section 6.2.5, les schémas de communications sur grilles de calcul doivent être conçus de manière à minimiser les communications les plus coûteuses, c'est-à-dire le plus souvent les communications de plus haut niveau hiérarchique. Les algorithmes utilisés pour les communications sur grilles de calcul doivent donc tenir compte de la topologie et pour chaque niveau de hiérarchie, minimiser le nombre de messages envoyés entre processus situés dans des groupes différents. Par exemple, un algorithme de diffusion utilisant un arbre binomial s'exécutant sur une grille de calcul constituée de plusieurs grappes envoie $O(log_2 N)$ messages inter-grappes, si N désigne le nombre de nœuds impliqués dans la diffusion. Un algorithme optimal pour cette grappe envoie $O(1)$ messages inter-grappes.

En suivant cette approche, on peut minimiser le nombre de communications de haut niveau hiérarchique, c'est-à-dire utiliser moins de messages plus le niveau de hiérarchie est haut, en utilisant la connaissance de la topologie transmise par l'environnement d'exécution. Un exemple illustre cette approche figure 6.7 avec quatre grappes, rouge, bleue, verte et rose. Un processus maître est élu dans chaque grappe pour effectuer les communications de niveau supérieur. Un algorithme optimisé diffuse le message entre les maîtres, puis les maîtres diffusent le message au sein de leur grappe en utilisant un algorithme optimisé dont ils sont la racine.

Cette stratégie permet de bénéficier des algorithmes de communications optimisés en admettant que chaque niveau de hiérarchie est homogène. Au regard des différences de performances de communications entre les niveaux de hiérarchie, on peut considérer que les performances de communications au sein d'un niveau sont relativement proches les unes des autres : elles ne diffèrent généralement pas de plus d'un ordre de grandeur, tandis qu'elles diffèrent de deux ou trois ordres de grandeur entre deux niveaux de hiérarchie. De plus, les performances des communications sur le réseau global pouvant subir de fortes variabilités dues à des communications concurrentes sur le réseau, une minimisation du nombre de communications sur le réseau global permet de limiter l'impact de ces variabilites sur les communications collectives.

6.3.3 Évaluation des performances

6.3.3.1 Conditions expérimentales

Les mesures de performances ont été effectuées sur deux plate-formes expérimentales : la grille expérimentale Grid'5000 [41] et la grappe QCG. Grid'5000 est présentée dans la section 2.2. Les expériences sur Grid'5000 ont pour but d'évaluer les performances des algorithmes dans un contexte de grappe de grappes. Nous avons utilisé un processus par machine, en mode dédié. De plus, nous avons utilisé seulement deux grappes (GdX à Orsay et Paravent à Rennes) afin de nous placer dans une situation extrême : en minimisant le nombre de communications entre les grappes, il ne devrait y avoir qu'une seule communication sur le réseau global. Nous avons vérifié le fait que les performances étaient similaires en ajoutant une troisième grappe (grappe Bordemer à Bordeaux).

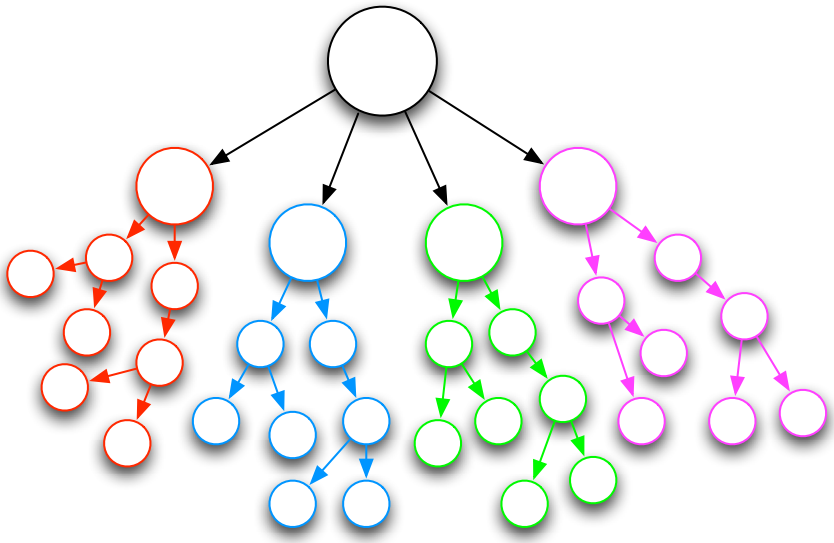


FIGURE *6.7 – Schéma de communication d'une diffusion hiérarchique : la racine de la diffusion envoie le paquet à un processus situé dans chaque grappe, qui sert de racine locale à une diffusion dans sa grappe, et ainsi de suite récursivement.*

Nous avons utilisé les grappes d'Orsay (grappe GdX) et Rennes (grappe Paravent). La plate-forme QCG est une grappe constituée de quatre nœuds de calcul et d'une frontale. Les nœuds sont équipés chacun d'un CPU double cœur Intel Pentium D cadencé à 2.8 GHz et disposant de 2x1Mo de cache de niveau deux. Ils sont interconnectés par un réseau Ethernet à 100 Mb/s. Les nœuds utilisent un noyau Linux 2.6.18.3 sur Grid'5000 et 2.6.22 sur la grappe QCG. La bande passante entre les grappes sur Grid'5000 est de 136,08 Mb/s et la latence est de 7,8 ms. La bande passante au sein des grappes est de 894,39 Mb/s et la latence est de 0,1 ms. Sur la grappe QCG, les communications par un segment de mémoire partagée offrent une bande passante de 3979,46 Mb/s et une latence de 0,02 ms, les communications TCP entre les nœuds ont une bande passante de 89,61 Mb/s (réseau 100 Mb/s) et une latence de 0.1 ms.

Nous avons évalué les performances de nos algorithmes en utilisant à chaque fois une hiérarchie à deux niveaux. La situation de grappe de grappes, représentée sur Grid'5000, permet d'obtenir un niveau de communications locales intra-grappe, et un niveau de communications globales inter-grappes. La situation de grappe de multi-processeurs est représentée par la grappe QCG et permet d'obtenir un niveau de communications sur une même machine et un niveau de communications intra-grappe.

Nous avons utilisé à chaque fois 32 processus. Sur Grid'5000, nous avons utilisé un processus par machine, et 16 processus sur chaque grappe. Sur la grappe QCG, nous avons utilisé 8 processus par machine. Les machines sont équipées de processeurs double cœur ; nous les avons donc sur-ordonnancées. Cependant, les programmes utilisés sollicitant les interfaces réseaux et effectuant très peu de calcul, on peut utiliser un nombre de processus supérieur au nombre de cœurs disponibles afin de solliciter les interfaces réseaux de manière plus intensive. Les processus ont été ordonnancés de manière séquentielle, c'est-à-dire que, par exemple sur Grid'5000, les processus de rang 0 à 15 ont été exécutés sur une grappe et les processus de rang 16 à 31 ont été exécutés sur l'autre grappe.

Mesurer le temps pris par une opération collective est un problème non trivial. On peut citer le projet SKaMPI [2], une suite de bancs d'essais spécialisée pour évaluer les performances des fonctions implémentant les opérations collectives en MPI. Cependant, les opérations collectives QCG ont été implémentées à l'extérieur de la bibliothèque MPI et sont appelées différemment des fonctions MPI. Par exemple, ces fonctions ont besoin de connaître la topologie. Leur prototype ne peut donc pas être le même que celui des fonctions MPI.

La méthode retenue ici est la même que celle utilisée par [110], par soucis de cohérence avec les travaux entrepris antérieurement sur le sujet. L'opération est précédée d'une barrière, qui assure une pseudo-synchronisation

2. Special Karlsruher MPI - benchmark (SKaMPI). `http://liinwww.ira.uka.de/~skampi/`

Listing 6.2 – Protocole de mesure de performances des opérations collectives

```
QCG_Barrier();
temps_initial = MPI_Wtime();
QCG_Opération_collective( ... );
QCG_Barrier();
temps_final = MPI_Wtime();
temps_passé = temps_final - temps_initial;
```

entre les processus. Le temps initial est pris à l'issue de cette barrière, puis l'opération collective est effectuée. La sémantique des opérations collectives MPI impose que les fonctions retournent dès que le processus courant a effectué sa part de l'opération. Il faut donc effectuer une nouvelle synchronisation des processus, au moyen d'une barrière. Le temps final est pris à la sortie de cette barrière. Le pseudo-code correspondant à ce protocole expérimental figure dans le listing 6.2. La barrière utilisée pour mesurer les opérations collectives hiérarchiques est présentée dans la section 6.3.3.3.

Les résultats suivants ont été publiés dans [59].

6.3.3.2 Implémentation et optimisation

Nous avons implémenté un prototype de nos communications collectives hiérarchiques en utilisant des fonctions MPI. Ainsi, nous avons pu bénéficier des opérations collectives déjà optimisées pour les environnements homogènes. Nous avons utilisé la bibliothèque de communications QCG-OMPI présentée au chapitre précédent et basée sur OpenMPI [85] 1.3a, et le composant `tuned` du framework implémentant les opérations collectives.

L'implémentation de certaines opérations collectives peut être optimisée en mettant en place un découpage des messages à partir d'une certaine taille de messages. Ce découpage a pour effet d'instaurer un pipeline entre les niveaux de hiérarchie et, dans le cas de communications sur une même machine, d'organiser un partage plus juste du bus système (partagé par tous les processus).

Les opérations ont été implémentées de manière à s'adapter à la topologie en obtenant la topologie et en définissant des communicateurs à chaque niveau. À chaque niveau, un communicateur est créé pour rassembler tous les processus partageant la même couleur à ce niveau. Un communicateur de niveau supérieur est créé entre les maîtres de ces communicateurs (en pratique, le processus de rang 0 sur chaque communicateur) pour permettre les communications de niveau supérieur.

L'idée de découper les messages en petits segments routés dans le réseau d'une machine parallèle a été exposée pour la première fois dans le protocole Wormhole [66], destiné aux machines massivement parallèles. L'impact de ce protocole, positif ou négatif, a été étudié dans [128].

La figure 6.8 présente les effets de ce découpage sur les performances d'une opération collective (MPI_Gather). Cette mesure a été effectuée sur la grappe QCG, en utilisant 32 processus (8 processus par machine). On constate qu'un découpage en huit portions est le meilleur choix à partir d'une certaine taille où les portions ne sont plus trop petites (environ 5 ko). Un découpage en un plus grand nombre de portions implique un pipeline trop long pour les deux niveaux de hiérarchie présents ici et n'utilise pas le bus système de façon optimale.

Ce découpage n'est pas utile pour les opérations déjà implémentées en mettant en place un pipeline, comme c'est le cas pour MPI_Reduce et MPI_Allreduce dans le composant d'OpenMPI que nous avons utilisé. Dans les autres cas, nous avons utilisé un découpage des messages en huit portions.

6.3.3.3 Barrière (MPI_Barrier)

La barrière hiérarchique utilisée respecte la sémantique imposée par la norme MPI qui impose qu'un processus ne sorte de la barrière qu'une fois que tous les processus y sont entrés. Pour cela, nous avons utilisé une barrière par niveau de hiérarchie, de bas en haut : on ne rentre dans la barrière de niveau supérieur qu'une fois que l'on est sorti de la barrière courante, et que tous les processus du groupe y sont entrés. Une fois le niveau maximum atteint, une diffusion envoie un message de taille nulle (en pratique, de un octet) à tous les processus. Les processus sortent de la barrière dès que ce message est reçu. La diffusion utilisée est la diffusion hiérarchique présentée dans la section 6.3.3.4. L'algorithme de barrière est exprimé en pseudo-code dans le listing 6.3.

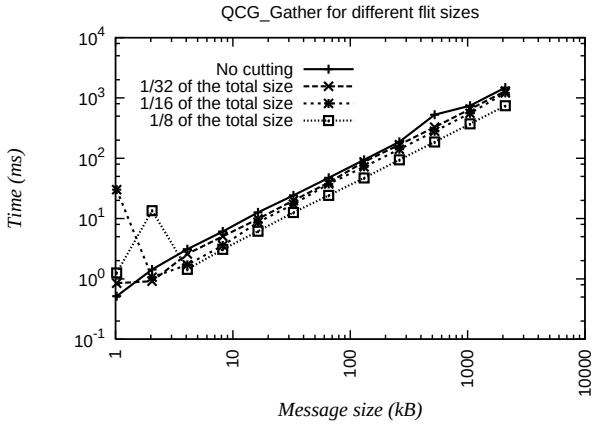


FIGURE *6.8 – Effet du découpage des messages et de la mise en pipeline des portions du message sur les performances d'une opération collective*

Listing 6.3 – Algorithme de barrière hiérarchique

```
/* les communicateurs sont dans un tableau de MPI_Comm */
QCG_Barrier( MPI_Comm **comms ) {
  for( i - depth[my_rank] - 1 ; i >= 0 ; i— ) {
    MPI_Barrier( comms[i] );
  }
  QCG_Bcast( &i, 1, MPI_INT, 0, comms );
}
```

La barrière hiérarchique n'effectue que des barrières entre processus de même niveau de hiérarchie. Les barrières effectuées à haut niveau de hiérarchie sont effectuées entre peu de processus seulement, limitant ainsi le nombre de messages coûteux. On a donc $O(2)$ messages inter-groupes à chaque niveau de hiérarchie, au lieu de $O(log_2 N)$ à $O(2N)$ (N étant le nombre de processus) avec un algorithme de barrière à plat.

6.3.3.4 Diffusion (MPI_Bcast)

L'algorithme de diffusion utilisé ici est proche de celui présenté dans [42] et dans [110]. Si la racine n'appartient pas au communicateur de plus haut niveau, elle l'envoie à un processus de ce communicateur. Le message est diffusé sur ce communicateur : il atteint alors les maîtres de tous les communicateurs de niveau inférieur en $O(logC)$ étapes, si C est le nombre de processus dans ce communicateur (typiquement, C au plus haut niveau est le nombre de grappes) et $O(1)$ messages inter-groupes. Chaque processus qui reçoit ce message le diffuse alors dans le communicateur de niveau inférieur, et ainsi de suite jusqu'au niveau le plus bas. L'algorithme de diffusion est exprimé en pseudo-code dans le listing 6.4.

Les performances de la diffusion hiérarchique ont été comparées avec celles de la diffusion à plat sur Grid'5000 (figure 6.9).

On constate des performances équivalentes pour les petits messages. La diffusion hiérarchique est plus stable (écart type plus petit) pour des messages de 64 ko : il s'agit du moment pour la bibliothèque MPI du passage entre le mode eager et le mode rendezvous. Le découpage des messages effectuées pour optimiser la collective hiérarchique permet de ne pas passer en mode rendezvous et de rester en mode eager : elle ne souffre donc pas de l'instabilité et de la chute des performances rencontrés par la diffusion à plat pour les messages de quelques centaines de kilooctets. Pour des messages de grande taille, on constate que la diffusion hiérarchique a de meilleurs performances (un ordre de grandeur d'écart).

La réduction du nombre de messages entre les groupes de processus (ici, entre les grappes) permet d'accélérer la diffusion en envoyant directement les messages à un processus de chaque grappe, qui se charge de le diffusion

Listing 6.4 – Algorithme de diffusion hiérarchique

```
/* les communicateurs sont dans un tableau de MPI_Comm */
QCG_Bcast( void* buffer, int nb_elements,
           MPI_Datatype dtype, int ROOT, MPI_Comm **comms ) {
  for( i = 0 ; i < depth[my_rank] ; i++ ) {
    MPI_Bcast( buffer, nb_elements, dtype, ROOT, comms[i] );
  }
}
```

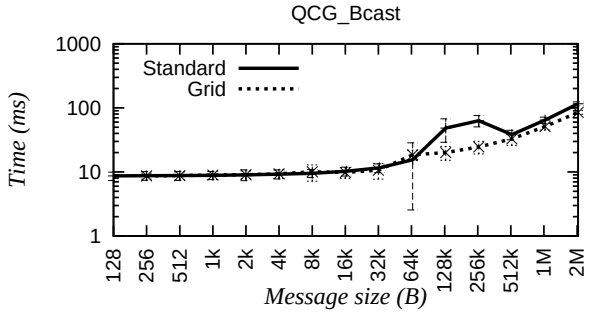


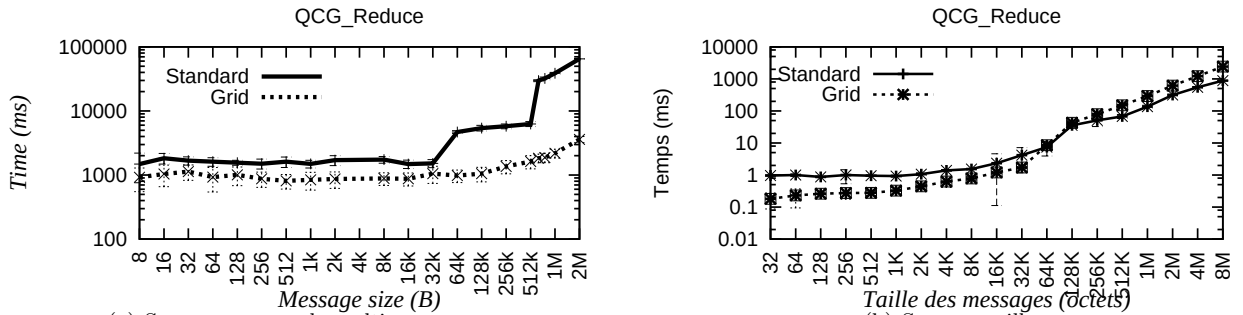
FIGURE *6.9 – Évaluation des performances d'une diffusion hiérarchique sur une grappe de multi-processeurs*

localement. Le fait de maximiser les communications locales et de minimiser les communications distantes (inter-grappes) apporte un gain plus important avec des messages de grande taille pour lesquels la bande passante est le critère limitant ; les communications locales ayant une bande passante plus importante, la diffusion hiérarchique bénéficie d'une maximisation de la bande passante disponible avec la minimisation du nombre de messages inter-grappes.

6.3.3.5 Réduction (MPI_Reduce)

Selon le même principe que pour la diffusion, la réduction hiérarchique est implémentée le long de la hiérarchie en effectuant des opérations sur les communicateurs de chaque niveau. Il est possible de décomposer l'opération de cette manière, car la norme MPI précise que l'opération effectuée par la réduction doit être commutative et associative. L'algorithme de diffusion est exprimé en pseudo-code dans le listing 6.5.

Les performances de la diffusion hiérarchique ont été comparées avec celles de la diffusion à plat sur la grappe QCG (figure 6.10(a)) et sur Grid'5000 (figure 6.10(b)).



(a) Sur une grappe de multi-processeurs

(b) Sur une grille

FIGURE *6.10 – Évaluation des performances d'une réduction hiérarchique.*

Listing 6.5 – Algorithme de réduction hiérarchique

```
/* les communicateurs sont dans un tableau de MPI_Comm */
QCG_Reduce( void* sbuf, void* rbuf, int nb_elements,
            MPI_Datatype dtype, MPI_Op op, int ROOT, MPI_Comm **comms ) {
  for( i = 0 ; i < depth[my_rank] ; i++ ) {
    MPI_Reduce( sbuf, rbuf, nb_elements, dtype, op, 0, comms[i] );
    sbuf = rbuf;
  }
  /* si la racine de la concaténation n est pas le rang 0
     du communicateur de plus haut niveau : on lui envoie
     le résultat
  */
  if( ( 0 == my_rank ) && ( 0 != root ) ) {
    MPI_Send( rbuf, rcount, rdtype, root, tag, MPI_COMM_WORLD );
  }
  if( root = my_rank ) {
    MPI_Recv( rbuf, rcount, rdtype, 0, tag, MPI_COMM_WORLD, &status );
  }
}
```

On constate que les performances de la réduction hiérarchique sont meilleures que celles de la réduction à plat sur la grappe de multi-cœurs et qu'elles s'améliorent pour des plus grandes tailles de messages. Cependant, on constate l'évolution inverse sur la grappe de grappes pour lesquelles les performances de la réduction à plat sont meilleures pour des grandes tailles de messages.

Les processus exécutés sur des machines multi-cœurs ou multi-processeurs sont confrontés au problème du partage du bus système. L'algorithme de réduction hiérarchique permet un meilleur partage du bus en ordonnançant les communications.

La réduction sur des messages de grande taille par rapport à la taille du communicateur sur lequel la réduction est effectuée est implémenté dans OpenMPI en utilisant un pipeline entre les processus. Les processus établissent une topologie de chaîne et se passent les portions de messages selon ce pipeline. La réduction à plat utilise tous les processus du communicateur en même temps, établissant alors un plus grand pipeline et bénéficiant alors d'une plus grande bande passante. La latence du pipeline (le temps de le remplir) est alors négligeable devant le gain en bande passante. Dans ces conditions, l'algorithme de réduction à plat présente de meilleures performances.

6.3.3.6 Réduction avec redistribution du résultat (MPI_Allreduce)

La réduction avec redistribution du résultat peut être implémentée de deux façons différentes et sémantiquement correctes :

- comme une réduction suivie d'une diffusion du résultat ;
- en utilisant un algorithme dédié (Bruck, etc)

Nous avons donc conçu un algorithme pour chacune de ces approches. Le premier utilise une réduction hiérarchique vers une racine présente dans le communicateur de plus haut niveau et une diffusion hiérarchique, comme présentés ci-avant. Le deuxième algorithme effectue une réduction avec redistribution du résultat vers le maître du groupe au plus bas niveau : tous les maîtres ont alors le résultat intermédiaire (local). Les maîtres effectuent une réduction avec redistribution du résultat au niveau supérieur, et ainsi de suite jusqu'au niveau le plus haut. Après la réduction avec redistribution du résultat au niveau le plus haut, tous les processus présents à ce niveau disposent du résultat final. Ils effectuent alors une distribution dans leur communicateur, et tous les processus recevant ce message le diffusent jusqu'au plus bas niveau de hiérarchie. Le deuxième algorithme de réduction avec diffusion du résultat est exprimé en pseudo-code dans le listing 6.6.

Listing 6.6 – Algorithme de réduction avec diffusion du résultat hiérarchique

```
/* les communicateurs sont dans un tableau de MPI_Comm */
QCG_Allreduce( void* sbuf, void* rbuf, int nb_elements,
               MPI_Datatype dtype, MPI_Op op, MPI_Comm **comms ) {
  for( i = depth[my_rank] - 1 ; i >= 0 ; i-- ) {
    MPI_Allreduce( sbuf, rbuf, nb_elements, dtype, op, comms[i] );
    sbuf = rbuf;
  }
  for( i = 1 ; i < depth[my_rank] ; i++ ) {
    MPI_Bcast( rbuf, nb_elements, dtype, ROOT, comms[i] );
  }
}
```

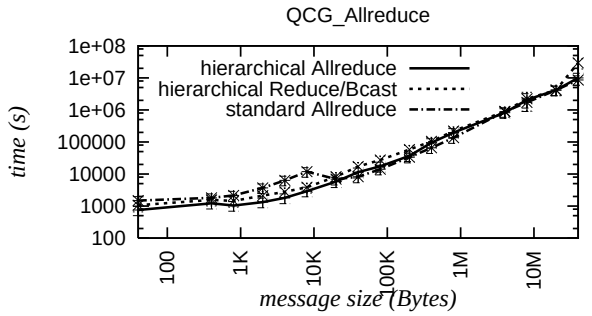


FIGURE *6.11 – Évaluation des performances d'une réduction avec redistribution du résultat hiérarchique sur une grappe de grappes.*

Les deux algorithmes de réduction avec redistribution du résultat ont été comparés sur la grappe QCG. Leurs performances sont représentées figure 6.11.

On constate que l'algorithme utilisant une réduction vers une racine puis une diffusion vers tous les processus est moins performant que le deuxième algorithme, et ce, pour toutes les tailles de messages. Pour des petits messages, où les temps de communications sont dominés par la latence, il est cependant plus performant que l'algorithme à plat.

La minimisation du nombre de messages dans les hauts niveaux de la hiérarchie (ici, les messages inter-machines) donne toujours de meilleures performances sur des petits messages. On subit $O(1)$ fois seulement la latence d'une communication à plus forte latence ce qui, pour des communications où la latence est le facteur limitant (petits messages), apporte toujours un gain par rapport à un algorithme impliquant $O(n)$ messages à forte latence (n désignant le nombre de processus).

L'algorithme de réduction suivi d'une diffusion présente d'inconvénient d'introduire une forme de synchronisation : la réduction doit être terminée pour que sa racine puisse initier la diffusion du résultat. Les processus situés à l'autre extrêmité de la toplogie de réduction et de diffusion sont donc inactifs pendant une certaine durée. À l'inverse, l'algorithme hiérarchisant une réduction avec redistribution du résultat permet de découper l'opération et de l'effectuer en parallèle sur plusieurs portions du système en même temps.

L'algorithme utilisant des réductions avec redistribution du résultat (deuxième algorithme) est plus performant, jusqu'à une certaine taille de messages où l'algorithme à plat devient le plus performant. Tout comme la diffusion, la réduction avec redistribution du résultat pour des messages de grande taille est implémentée dans OpenMPI en mettant en place une topologie d'anneau et en établissant un pipeline, profitant ainsi d'une bande passante maximisée.

Listing 6.7 – Algorithme de concaténation hiérarchique

```
/* les communicateurs sont dans un tableau de MPI_Comm
   et on a les déplacements à faire à chaque niveau de profondeur
   dans un tableau d entiers
 */
QCG_Gather( void* sbuf, int scount, MPI_Datatype sdtype,
            void* rbuf, int rcount, MPI_Datatype rdtype,
            int root, MPI_Comm* comms ){
  MPI_Gather( sbuf, scount, sdtype,
              rbuf, rcount, rdtype,
              root, comms[depth[my_rank]-1] );
  for( i = depth[my_rank] - 2 ; i >= 0 ; i-- ) {
    sbuf = rbuf;
    scount = rcount;
    MPI_Gatherv( sbuf, scount, sdtype,
                 rbuf, rcount, displ[i], rdtype,
                 0, comms[i] );

  }
  /* si la racine de la concaténation n est pas le rang 0
     du communicateur de plus haut niveau : on lui envoie
     le résultat
  */
  if( ( 0 == my_rank ) && ( 0 != root ) ) {
    MPI_Send( rbuf, rcount, rdtype, root, tag, MPI_COMM_WORLD );
  }
  if( root = my_rank ) {
    MPI_Recv( rbuf, rcount, rdtype, 0, tag, MPI_COMM_WORLD, &status );
  }
}
```

6.3.3.7 Concaténation (MPI_Gather)

La concaténation de messages peut se faire de manière hiérarchique à condition que les processus soient répartis de manière continue dans les groupes. Une concaténation est effectuée sur les communicateurs de plus bas niveau vers un processus maître, puis les maîtres effectuent une concaténation des tampons résultats sur le communicateur de niveau supérieur, et ainsi de suite jusqu'à la racine de l'opération. Il faut remarquer que si les communicateurs d'un niveau donné ne sont pas tous de la même taille, les tampons résultant des concaténations ne seront pas de la même taille. Il sera alors nécessaire d'utiliser au niveau supérieur une opération permettant d'utiliser des tampons de taille variable (MPI_Allgatherv). Cet algorithme permet d'effecteur des concaténations locales et de n'envoyer qu'un seul message sur le réseau de niveau supérieur. On gagne en latence et la bande passante est mieux remplie. L'algorithme de concaténation de messages hiérarchique est exprimé en pseudo-code dans le listing 6.7.

Les performances de l'algorithme de concaténation de messages sont comparées avec celles de la concaténation de messages à plat sur la grappe de multi-processeurs QCG et présentées figure 6.12.

On constate que, quelle que soit la taille des messages, l'algorithme hiérarchique présente de meilleures performances avec un écart d'environ un ordre de grandeur. La concaténation effectuée localement en communiquant au moyen d'un segment de mémoire partagée permet de n'effectuer qu'une seule communication sur le réseau de la grappe. Cet algorithme n'envoyant que $O(1)$ messages sur les réseaux à latence plus importante, le gain est plus important pour des messages de petite taille, où la communication est limitée par la latence.

Le gain en performances est notable aussi pour des messages de taille importante. La topologie utilisée pour

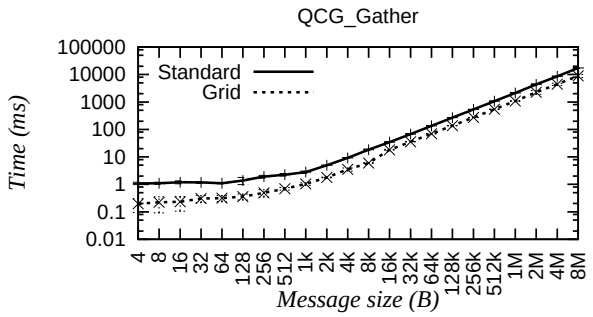


FIGURE *6.12 – Évaluation des performances d'une réduction concaténation de messages sur une grappe de multi-processeurs*

effectuer la communication à chaque niveau de hiérarchie est plus équilibrée que pour une diffusion à plat et la bande passante disponible à chaque niveau est mieux utilisée.

6.3.4 Conclusion sur les opérations collectives hiérarchiques

Dans cette section, on a vu des algorithmes permettant de hiérarchiser des opérations collectives afin de les adapter à la topologie logique de l'application, laquelle conditionne la topologie physique des ressources utilisées. Le prinicpe de base de ces algorithmes consiste à ordonnancer les communications en fonction d'un schéma hiérarchique pour minimiser le nombre de communications à haut niveau de hiérarchie et homogénéiser autant que possible les communications à un niveau donné. Ces algorithmes se basent sur des algorithmes existants, déjà optimisés.

L'étude des performances de ces communications collectives hiérarchiques a été effectuée pour deux types de systèmes à deux niveaux courants : les grappes de multi-processeurs et les grappes de grappes, effectuant une seule transition à la fois. On a pu voir que cette hiérarchisation apporte un gain important en performances pour des communications collectives basées sur des topologies de communications en arbres, comme la diffusion ou la concaténation. Les opérations collectives basées sur des topologies qui maximisent la bande passante en établissant un pipeline, pour recouvrir les communications avec une opération (opérations de réduction) présentent moins d'intérêt à la hiérarchisation pour des messages de grande taille, où la bande passante est importante, pour des communications réseaux. Lorsque plusieurs processus sont exécutés sur chaque machine (grappe de multi-processeurs), la hiérarchisation permet une meilleure utilisation du bus système. Enfin, la hiérarchisation associée à un pré-découpage des messages permet d'établir un pipeline des communications à travers les niveaux successifs de hiérarchie.

6.4 Adaptation des schémas de calcul et communications

En utilisant la connaissance a priori des ressources qui seront utilisées, les applications peuvent être adaptées à la topologie sur laquelle elles sont exécutées. Deux exemples d'applications typiques utilisant des schémas de calcul et de communications courants sont présentés dans cette section : le schéma maître-esclave est présenté section 6.4.1 et un exemple de décomposition de domaine hiérarchique pour l'algèbre linéaire est présenté section 6.4.2.

6.4.1 Modèle maître-esclave

6.4.1.1 Algorithme

Le schéma maître-esclave est très répandu pour sa simplicité de mise en place et ses propriétés d'équilibrage de charges et de résilience pour des applications de type "*sac de tâches*" : le calcul peut être découpé en tâches

indépendantes les unes des autres par un processus dit *maître* qui distribue les tâches à des processus dits *esclaves*[3].

Il n'est cependant pas adapté à des exécutions à grande échelle, du fait du point central de congestion constitué par le maître. Si le nombre d'esclaves est trop important, le maître n'a pas le temps de traiter les résultats qu'il reçoit et de renvoyer un nouveau jeu de données à traiter dans le temps dont il dispose pour chaque processus esclave. Les esclaves qui rapportent leurs résultats sont alors mis en file d'attente où ils perdent du temps et ne calculent pas.

De plus, sur une grille de calcul où les temps de communications sont différents selon si l'on sort ou non d'une entité physique de la grille (grappe, machine, domaine d'administration...), les temps nécessaires pour envoyer les résultats et recevoir un nouveau jeu de données dépend de la localisation géographique de l'esclave par rapport au maître : s'ils sont proches l'un de l'autre, ce délai est court, mais s'ils sont situés dans des domaines d'administration différents, le temps durant lequel l'esclave doit communiquer avec le maître correspond à un temps durant lequel il ne participe pas au calcul.

Une solution est de ne pas utiliser une seule file d'attente mais plusieurs, en utilisant plusieurs maîtres qui gèrent chacun un ensemble d'esclaves. Ce schéma peut être hiérarchisé, en ayant une file d'attente globale gérée par un maître de plus haut niveau, dit *super-maître*. Le super-maître découpe les données du problème et les envoie à des processus de niveau inférieur, comme un maître.

Aux niveaux intermédiaires sont introduits des processus d'un type nouveau : les *contremaîtres*. Ils reçoivent un jeu de données du niveau de hiérarchie supérieur, le découpent et le distribuent entre les processus de leur groupe, collectent les résultats et les transfèrent au niveau supérieur. Les contremaîtres peuvent communiquer avec des esclaves ou avec d'autres contremaîtres. Dans les deux cas, leur rôle est le même : ils demandent au niveau inférieur de traiter des données. En quelque sorte, les contremaîtres sont vus par leur niveau supérieur comme des esclaves.

La figure 6.13 présente un cas de schéma maître-esclave à plusieurs niveaux de hiérarchie. Le super-maître voit trois contremaîtres. Deux d'entre eux distribuent les données à traiter directement à des esclaves, tandis que celui du milieu les distribue à des contremaîtres de niveau inférieur qui seront chargés de distribuer aux esclaves.

Les contremaîtres sont des composants locaux à leurs esclaves : ils sont situés dans le même groupe de plus bas niveau. Ainsi, les communications étant d'autant plus fréquentes qu'elles ont lieu à un niveau bas de la hiérarchie, on utilise des moyens de communications rapides et à faible latence pour les communications qui ont lieu souvent tandis que les communications à plus forte latence ont lieu moins souvent, pour communiquer un jeu de données ou de résultats correspondant au calcul de plusieurs esclaves.

Les algorithmes de chacun des composants (super-maître, contremaître et esclave) sont décrits en pseudo-code par le listing 6.8. L'algorithme d'un schéma maître-esclave classique est décrit par le listing 6.9.

6.4.1.2 Une application : Ray2mesh

Ray2mesh [100] est une application de géophysique appliquant la loi de Snell-Descartes à des rayons sismiques mesurés et appliqués à une représentation du terrain. Elle est constituée de trois phases : une phase de communications collectives, permettant de diffuser les données du problème, une phase de calcul selon un schéma maître-esclave et une phase de communications où les résultats sont recombinés. L'ordre dans lequel les rayons sont calculés n'a pas d'importance.

Nous avons adapté l'application à la topologie multi-grappes de la grille que nous avons utilisée. Pour cela, nous avons utilisé des opérations collectives adaptées présentées ci-avant (diffusions) et un schéma maître-esclave hiérarchique à un niveau de contremaîtres. Cette version adaptée pour la grille et ses performances ont été publiés dans [59, 58, 64].

Les performances ont été évaluées sur Grid'5000 en utilisant les grappes de Orsay (GdX), Bordeaux (Bordemer) et Rennes (Paravent) en respectant la répartition suivante : deux processus à Orsay pour un processus à Bordeaux et un processus à Rennes, dans la limite du nombre de processeurs disponibles à Bordeaux (32) au-delà de laquelle nous avons uniquement augmenté le nombre de processeurs à Orsay.

Nous avons mesuré de manière progressive l'adaptation à la topologie : la première étape, la plus automatique, consiste à remplacer les opérations collectives pour grappes par des opérations collectives hiérarchiques. La

3. La langue française n'a pas d'équivalent au plus politiquement correct "*master-worker*" anglophone

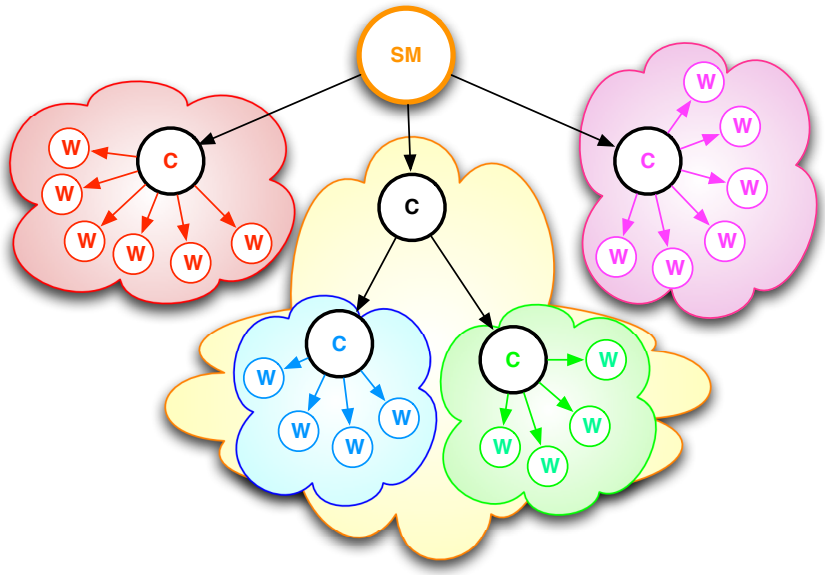


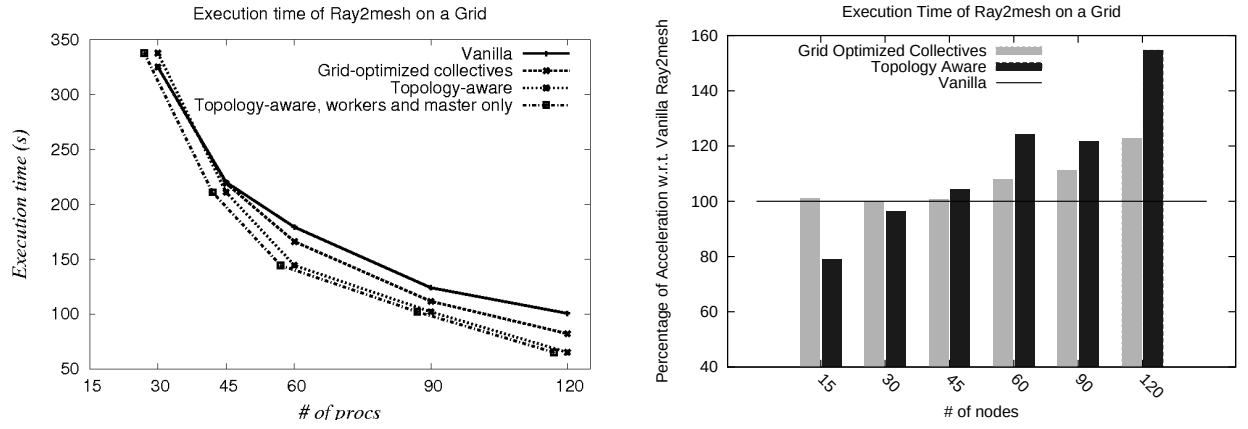
FIGURE *6.13 – Un schéma de calcul suivant le modèle maître-esclave disposant de plusieurs files d'attente : une file d'attente de haut niveau (notée SM, comme Super Maître) distribue les données à des files d'attente locales (notées C, comme Contremaîtres). Les contremaîtres distribuent les données entre leurs esclaves, ou à des contremaîtres de plus bas niveau.*

Listing 6.8 – Algorithme d'un schéma maître-esclave hiérarchique

```
procédure maître-esclave-hiérarchique( données ) {
  if( je_suis_le_super-maître ){
    while ( données ) {
      distribuer_aux_contremaîtres( portion_de_données );
      recevoir_des_contremaîtres( portion_de_résultat );
    }
  } else {
    if( je_suis_un_contremaître ) {
      while( !FIN ){
        recevoir_donnée_du_super-maître_ou_du_contremaître( donnée );
        while( donnée ) {
          distribuer_aux_eclaves_ou_contremaîtres( portion_de_donnée );
          recevoir_des_esclaves_ou_contremaîtres( portion_de_résultat );
        }
        envoyer_au_super-maître_ou_au_contremaître( résultat );
      }
    } else { /* esclave */
      while( !FIN ) {
        recevoir_donnée_du_contremaître( donnée );
        envoyer_au_contremaître( résultat );
      }
    }
  }
}
```

Listing 6.9 – Algorithme d'un schéma maître-esclave classique

```
procédure maître-esclave-hiérarchique( données ) {
  if( je_suis_le_maître ){
    while ( données ) {
      distribuer_aux_esclaves( portion_de_données );
      recevoir_des_esclaves( portion_de_résultat );
    }
  } else { /* esclave */
    while( !FIN ) {
      recevoir_donnée_du_maître( donnée );
      envoyer_au_maître( résultat );
    }
  }
}
```



(a) Temps absolus (b) Temps relatifs

FIGURE *6.14 – Comparaison du gain de temps d'exécution obtenu par les trois niveaux d'adaptation de Ray2mesh pour la grille : pas d'adaptation, utilisation d'opérations collectives adaptées, schéma maître-esclave hiérarchique sur un nombre de processus égal à celui de la version non adaptée, et en rajoutant des processus pour exécuter les contremaîtres*

deuxième étape consiste à transformer le schéma maître-esclave pour le rendre hiérarchique. Enfin, la dernière étape consiste à ajouter des processus au nombre total de processus, pour prendre en compte les contremaîtres et garder un nombre d'esclaves constant. Les performances comparées de ces trois étapes d'adaptation à la grille ainsi que l'application non adaptée sont représentées figure 6.14(a) avec un écart type inférieur à 1%.

La courbe en trait continu donne les performances de l'application non modifiée. Juste en-dessous, la ligne en traits interrompus montre les performances de Ray2mesh utilisant des opérations collectives optimisées pour la grille. Il s'agit du premier niveau d'adaptation à la grille : l'application n'est pas modifiée, mais elle utilise des opérations collectives adaptées à la topologie. On constate une amélioration des performances de 9,5% pour 180 processus, tandis que les performances sont les mêmes à petite échelle.

La courbe en pointillés présente les performances de la version hiérarchique de Ray2mesh. Il s'agit de l'étape suivante de l'adaptation de l'application à la grille : les communications, qu'elles fassent partie d'une opération collective ou de l'application elle-même, sont adaptées à la topologie de la grille. On constate un gain d'environ 20% par rapport à la version non adaptée pour 180 processus, mais l'application est plus lente à petite échelle.

La hiérarchisation du schéma maître-esclave introduit les contremaîtres, un nouveau type de processus ne participant pas directement au calcul. À nombre de processus égal, on a donc un nombre d'esclaves inférieur. En ajoutant des processus au nombre total de processus afin d'atteindre un nombre d'esclaves égal à celui

obtenu avec un schéma maître-esclave traditionnel, on obtient la quatrième courbe. On a alors des performances similaires à petite échelle et un gain d'environ 35% pour 180 processus.

Ces résultats sont présentés sous une autre forme sur la figure 6.14(b) : on a alors des accélérations relatives au temps d'exécution de la version non modifiée de Ray2mesh, représenté par la ligne horizontale. Les rectangles gris représentent l'accélération obtenue par l'application non hiérarchisée, mais utilisant les opérations collectives optimisées. On constate que cette version n'est jamais plus lente que la version non adaptée à la grille. L'adaptation des opérations collectives à la grille est donc une première étape, facile à mettre en place, et ne pénalisant jamais l'exécution quelle que soit l'échelle à laquelle on l'exécute. Cependant, pour aller plus loin et à plus grande échelle, l'adaptation des schémas de calcul et de communication de l'application est nécessaire.

6.4.2 Algèbre linéaire

Passer à l'échelle de la grille sur des applications d'algèbre linéaires reste un véritable défi [132]. Dans cette section, nous étudions le passage à l'échelle en termes de performance d'une opération centrale d'algèbre linéaire, la décomposition (ou factorisation) d'une matrice dense en un produit QR d'une matrice orthogonale Q et d'une matrice orthogonale supérieure R. Basée sur des transformations orthogonales, cette méthode est réputée pour sa stabilité numérique et est, entre autres, une méthode très utilisée pour la résolution de problèmes aux moindres carrés [94]. Nous restreignons notre étude à la factorisation de matrices hautes et fines (dites *tall and skinny*) et vérifiant $M \gg N$ si M est le nombre de lignes et N le nombre de colonnes) dont nous présentons les applications les plus importantes dans la section 6.4.2.1. L'originalité de l'approche présentée ici consiste à utiliser les fonctionnalités topologiques de QCG-OMPI pour implémenter et exécuter un algorithme (présenté en section 6.4.2.2) assurant une localité des communications (en effectuant la plupart des communications au sein de sous-groupes de processus) afin de limiter les communications inter-sites et ainsi passer à l'échelle (section 6.4.2.4).

6.4.2.1 Motivations pour la factorisation QR de matrices hautes et fines

La factorisation de matrices denses, hautes et fines est utilisée directement comme noyau de calcul de plusieurs applications importantes d'algèbre linéaire présentées succintement ici et détaillées plus précisément dans [68]. Les méthodes itératives par bloc par exemple effectuent fréquemment une telle factorisation, aussi bien pour la résolution de systèmes linéaires à seconds membres multiples de type $AX = B$ dans des variantes de GMRES, QMR ou CG, que pour la résolution de problèmes aux valeurs propres. Une formulation des méthodes de Krylov, appelée *s-step Krylov methods* utilise une factorisation QR pour orthogonaliser les vecteurs de la base du sous-espace de Krylov. Enfin, la factorisation QR d'une matrice haute et fine intervient à chaque factorisation d'un panneau (un bloc de colonnes) de la décomposition QR d'une matrice quelconque. Parvenir à passer à l'échelle de la grille cette méthode représente donc un enjeu important.

6.4.2.2 Algorithmes de factorisation QR à évitement de communications (TSQR et CAQR)

Les algorithmes implémentés dans les bibliothèques standards d'algèbre linéaire pour machines à mémoire distribuée telles que ScaLAPACK [52] et basés sur une interface MPI peuvent être utilisés tels quels sur la grille en utilisant une bibliothèque de communications MPI, comme par exemple QCG-OMPI. Néamoins, étant optimisées pour des grappes de calculs, ces méthodes ne sont pas adaptées à la topologie des grilles en grappe de grappes. Les latences inter-sites étant trop coûteuses, les études expérimentales effectuées jusqu'à présent dont nous avons connaissance ont principalement eu pour objectif de traiter des problèmes de plus grande taille grâce à la mémoire distribuée sur les différents sites géographiques comme dans le cadre du projet GrADS [133] [4]. Pour les matrices de taille permettant d'être traitées sur un site donné, en revanche, ces études ont montré qu'une performance plus importante était obtenue lorsque les calculs étaient menés sur un seul site que lors de calculs distribués sur plusieurs sites géographiques distincts. Notons enfin qu'adapter la charge pour tenir en compte de l'hétérégonéité de la puissance de calcul des processeurs [144] [5] est un problème orthogonal, puisque le problème traité ici est lié à la hiérarchie des liens de communications.

4. Software Support for High-Level Grid Application Development `http://www.hipersoft.rice.edu/grads/`
5. `http://hcl.ucd.ie/project/HeteroScaLAPACK`

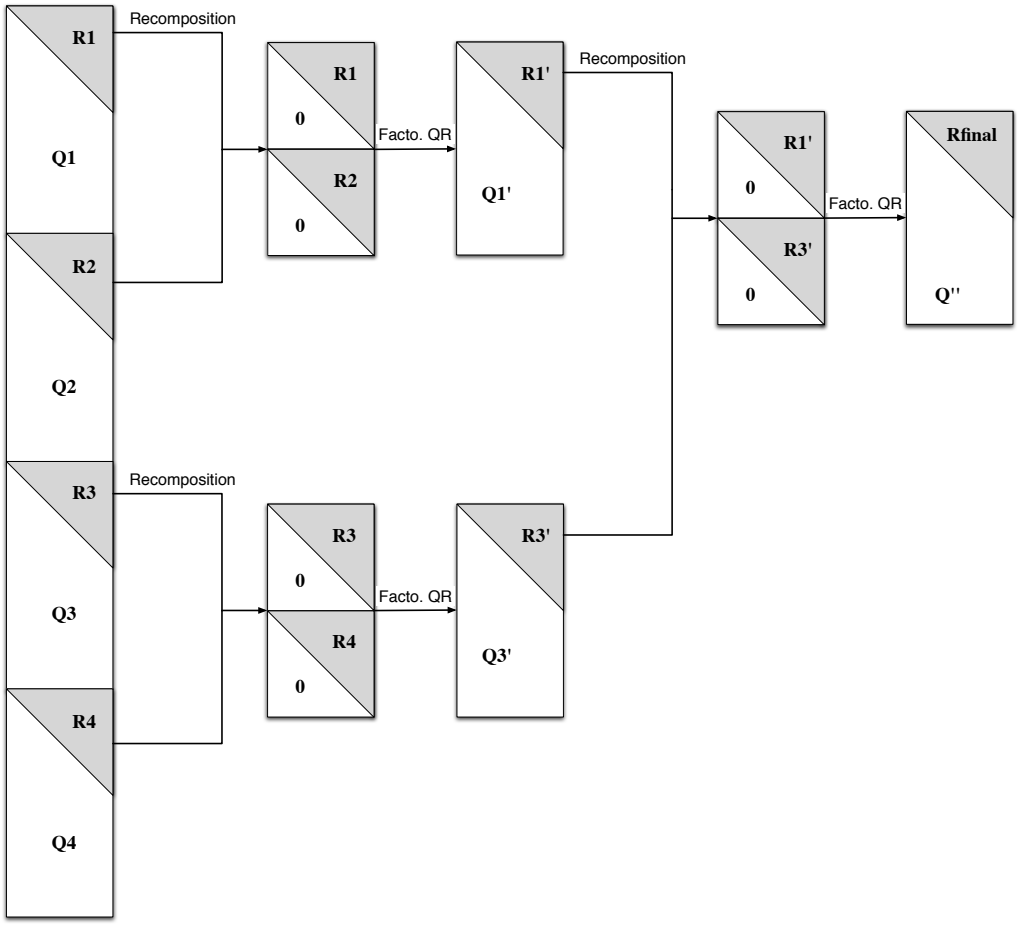


FIGURE *6.15 – Algorithme TSQR. La matrice initiale, haute et fine, est découpée en quatre domaines factorisés indépendamment (à gauche). Les facteurs triangulaires obtenus sont recomposés deux par deux et factorisés (milieu) jusqu'à obtenir le facteur triangulaire final R_{final} (droite).*

L'algorithme TSQR [68] (lui-même basé sur [139]), pour *Tall and Skinny QR* consiste à décomposer verticalement une matrice haute et fine en sous-matrices ou *domaines* (de tailles égales dans notre cas) et à limiter les communications entre chacun de ces domaines. Cet algorithme est donc particulièrement adapté à l'environnement de grille. En effet, en s'appuyant sur un intergiciel (QCG-OMPI) permettant d'obtenir des ressources matérielles correspondant aux besoins algorithmiques, il devient alors possible d'associer différents sites géographiques à différents domaines et ainsi limiter les communications inter-sites, ce qui est essentiel pour la performance. L'algorithme TSQR est illustré par la figure 6.15. La première étape consiste à effectuer une factorisation QR locale à chaque domaine, comme indiqué sur la colonne de gauche de la figure 6.15. Les facteurs triangulaires R_i locaux ainsi obtenus sont recomposés deux par deux : ici R_1 et R_2 d'une part, et R_3 et R_4 d'autre part. Chacune des ses paires (dites "triangle au dessus d'un triangle") sont alors factorisées à leur tour, et un nouveau facteur R'_i est issu de cette factorisation. Les matrices résultantes sont à nouveau recomposées, jusqu'à obtenir le facteur triangulaire final R_{final}. L'ensemble de ces "couplages-factorisations" est appelé *réduction* et forme un arbre binaire dont les arêtes correspondent aux couplages et les noeuds aux factorisations.

L'algorithme TSQR peut être à son tour utilisé pour la factorisation des panneaux (blocs-colonnes) d'un algorithme effectuant une factorisation QR de matrices générales, appelé *Communication Avoiding QR* ou *CAQR* [68]. Enfin, le volume de communication généré par TSQR et CAQR entre les différents domaines est minimal (sous certaines conditions, voir [68] pour davantage de détails). Transposé à une exécution sur la grille sur une matrice d'ordre suffisamment grand, cela signifie que TSQR (et CAQR) génèrent un volume

rang	0	1	2	3	4	5	6	7
profondeur	3	3	3	3	3	3	3	3
couleurs	W	W	W	W	W	W	W	W
	C1”	C1”	C1”	C1”	C3”	C3”	C3”	C3”
	C1’	C1’	C2’	C2’	C3’	C3’	C4’	C4’

FIGURE *6.16 – Tableau de couleurs pour une factorisation QR à évitement de communications sur 4 groupes de processus.*

de communication inter-sites minimal parmi l'ensemble des algorithmes de factorisations QR basés sur des transformations orthogonales et équilibrant la charge entre les différents sites.

6.4.2.3 Implémentation

Nous avons implémenté l'algorithme TSQR en utilisant QCG-OMPI et les routines d'algèbre linéaire fournies par ScaLAPACK [52] et les BLAS [53] implémentés dans GotoBLAS [95].

Les sous-domaines considérés par TSQR correspondent à des domaines logiques du point de vue de QCG-OMPI. On bénéficie alors pleinement du mécanisme d'évitement de communications de l'algorithme : les factorisations QR, qui effectuent des phases de communications synchrones et relativement nombreuses, sont effectuées entre des processus considérés comme “proches”. Il peut s'agir de machines sur une même grappe dans le cas d'un calcul sur grille, ou de cœurs situés sur une même machine dans le cas d'un calcul sur grappe de multiprocesseurs.

QCG-OMPI permet de définir, grâce au mécanisme d'ordonnancement décrit dans la section 6.2, des groupes de processus associés à des impératifs matériels ou de performances. Ainsi, en définissant des groupes de processus et en spécifiant les performances de communications au sein de ces groupes, et entre ces processus (principalement une latence faible entre processus d'un même groupe), on assure un ordonnancement des processus sur la grille permettant de tirer parti efficacement de l'algorithme TSQR.

Les factorisations locales QR (gauche de la figure 6.15) sont effectuées directement par des appels à ScaLAPACK, en utilisant une grille de communication basée sur les processeurs d'un même groupe, correspondant à une même couleur dans la dernière ligne de la table 6.16. Ainsi est utilisé un communicateur par domaine, construit grâce à la topologie obtenue par QCG-OMPI. En pratique, QCG-OMPI assurera que des processeurs appartenant à un même groupe se trouvent sur un même site géographique (une même grappe de calcul).

Les couplages de matrices sont ensuite effectuées entre processus de différents domaines. Une fois déterminé le domaine avec lequel le couplage va être effectuée, chaque processus qui ne va pas participer au calcul envoie sa partie du résultat au processus du domaine qui va effectuer le calcul de même rang dans son communicateur local. Par exemple, à la première étape, les processus du domaine 0 reçoivent les données des processus du domaine 1, les communications se faisant entre processus de même rang dans leur communicateur local respectif.

Les factorisations QR sur les matrices ainsi couplées sont à nouveau effectuées par des appels à ScaLAPACK sur les communicateurs de niveau supérieur. Pour l'exemple donné sur la figure 6.15 sur huit processus, le tableau de couleurs est donné par le tableau 6.16. Afin d'éviter d'effectuer des opérations sur des éléments nuls lors de la factorisation des matrices couplées, leur structure particulière (superposition de deux matrices triangulaires supérieures) pourrait être prise en compte. Néamoins, pour les matrices que nous étudions, suffisamment hautes et fines, la quantité de calcul due aux factorisations intermédiaires de matrices couplées est faible par comparaison avec la quantité de calcul effectuée lors des factorisations locales initiales. Une telle optimisation n'est donc pas critique. Nous remplissons donc de zéros explicites les parties triangulaires inférieures (qui contenaient initialement les vecteurs de Householder permettant le calcul de la matrice Q) de chaque matrice couplée. Il est alors possible de factoriser chaque couple de matrice formé avec un seul appel à une factorisation QR standard de ScaLAPACK sur une matrice rectangulaire. Notre prototype construit ainsi les facteurs triangulaires intermédiaires R_i. Les processeurs qui ont effectué la factorisation QR du domaine 1 effectuent celle de la matrice formée par R_1 et R_2, et ceux qui ont effectué la factorisation QR du domaine 3 effectuent celle de la matrice formée par R_3 et R_4. Finalement, le facteur triangulaire R_{final} obtenu est numériquement correct ; en revanche, l'information nécessaire à la construction de la matrice orthogonale Q (les vecteurs de Householder) est écrasée.

Enfin, si on note P le nombre de domaines utilisés, N le nombre de colonnes de la matrice et M le nombre de lignes, notre implémentation ne prend en charge que des matrices vérifiant $\frac{M}{P} \geq N$.

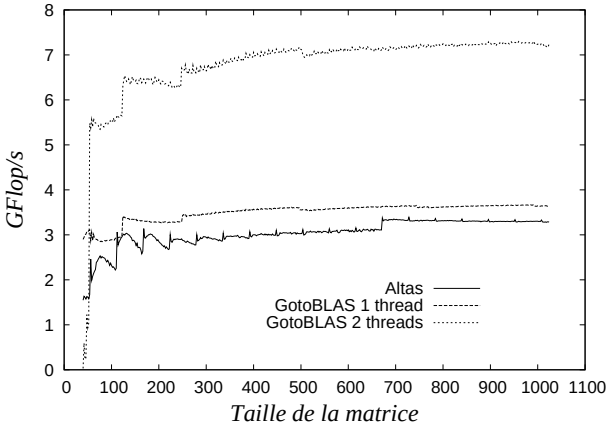


FIGURE *6.17 – Performances obtenues par une multiplication de matrices séquentielle en utilisant GotoBLAS.*

6.4.2.4 Résultats expérimentaux

Environnement de mesures

Nous avons étudié la performance de notre implémentation de TSQR sur Grid'5000 en utilisant les grappes GdX, Bordereau, Pastel et Helios, décrites dans la section 2.2 du chapitre 2. Les fréquences des micro-processeurs équipant les machines ne sont pas identiques selon les grappes ; les factorisations QR impliquant une importante synchronisation entre les processus, la rapidité du calcul sera minoré par les machines les moins rapides. La grappe équipée des micro-processeurs les plus lents est GdX : les mesures sur une seule grappe ont donc été effectées sur celle-ci, pour des raisons d'équité entre les mesures.

Nous avons utilisé les bibliothèques de calcul et de communications suivantes :

- ScaLAPACK 1.8.0 ;
- BLACS MPI 1.1 ;
- GotoBLAS 1.26.

Les BLAS étant les briques de bases effectuant la plupart des calculs, leur propre performance est critique pour la performance globale de l'application. Il est donc essentiel de se reposer sur une implémentation efficace. Nous avons comparé les performances d'ATLAS [163] et de GotoBLAS sur une machine de GdX équipée de deux micro-processeurs cadencés à 2.0 GHz en effectuant une multiplication matrice-matrice séquentielle (opération appelée DGEMM dans la terminologie LAPACK). Les performances obtenues sont rapportées dans la figure 6.17. GotoBLAS peremet de spécifier le nombre de threads à utiliser : comme la machine utilisée est équipée de deux micro-processeurs, nous avons mesuré la performance obtenue avec un thread et deux threads. Même en utilisant un seul thread, GotoBLAS présente de meilleures performances : nous l'utiliserons dans la suite. Afin de limiter coût de partage de l'accès à la carte réseau, nous n'utilisons qu'un processus par machine. Nous pouvons alors utiliser les deux processeurs de la machine grâce aux BLAS multithreadés. Dans nos expériences, nous présenterons alternativement des résultats avec la version mono-thread (un processeur sur deux n'est alors pas utilisé) de GotoBLAS et avec la version utilisant deux threads par processus (tous les processeurs sont alors utilisés).

Le nombre d'opérations effectuées par une factorisation QR sont données par la formule :

$$flops = 2M \cdot N^2 - \frac{2}{3}n^3 \tag{6.1}$$

La performance est obtenue en divisant le nombre d'opérations par le temps de calcul.

Choix du nombre de domaines intra-site

On peut faire varier le nombre de domaines exécutés dans une même grappe : on a alors une topologie logique composée d'un nombre de groupes de processus supérieur au nombre de grappes.

La figure 6.18 présente une comparaison de la performance obtenue sur une seule grappe, en utilisant des matrices de quatre blocs de large ($n = 256$) et en variant le nombre de domaines utilisés. Dans le cas d'une ma-

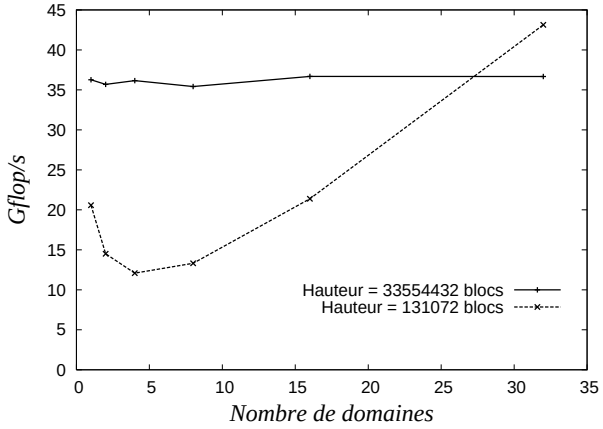


FIGURE *6.18 – Performance de TSQR sur une grappe en fonction du nombre de domaines en utilisant deux threads par machine.*

trice de hauteur limitée ($M = 131\ 072$), la performance est significativement améliorée en utilisant un domaine par processus (triplement de la performance). En effet, pour une telle matrice, les temps de communications sont importants par rapport aux temps de calcul locaux. Les communications déterminent alors la performance globale. Le schéma TSQR, limitant leur quantité, est alors essentiel à appliquer en associant un domaine à un processeur. En revanche, pour une matrice de hauteur plus importante ($M = 33\ 554\ 432$), les temps de calculs dominent largement les communications si bien que le nombre de domaines en lequel une matrice est découpée n'influe pas sur la performance, si la factorisation effectuée sur une grappe de calcul donnée.

En extrapolant ces résultats sur un calcul inter-sites (plusieurs grappes géographiquement distribuées), on peut supposer que pour des matrices suffisamment hautes, le nombre de domaine par site n'influe pas sur la performance. Si les résultats que nous reportons dans la suite pour l'algorithme TSQR correspondent à la performance optimale obtenue en fonction du nombre de domaines par site, en pratique nous avons bien constaté que le nombre de domaines par site influe peu. Nous ne reviendrons donc pas sur l'ajustement du nombre de domaines par site (il importe peu).

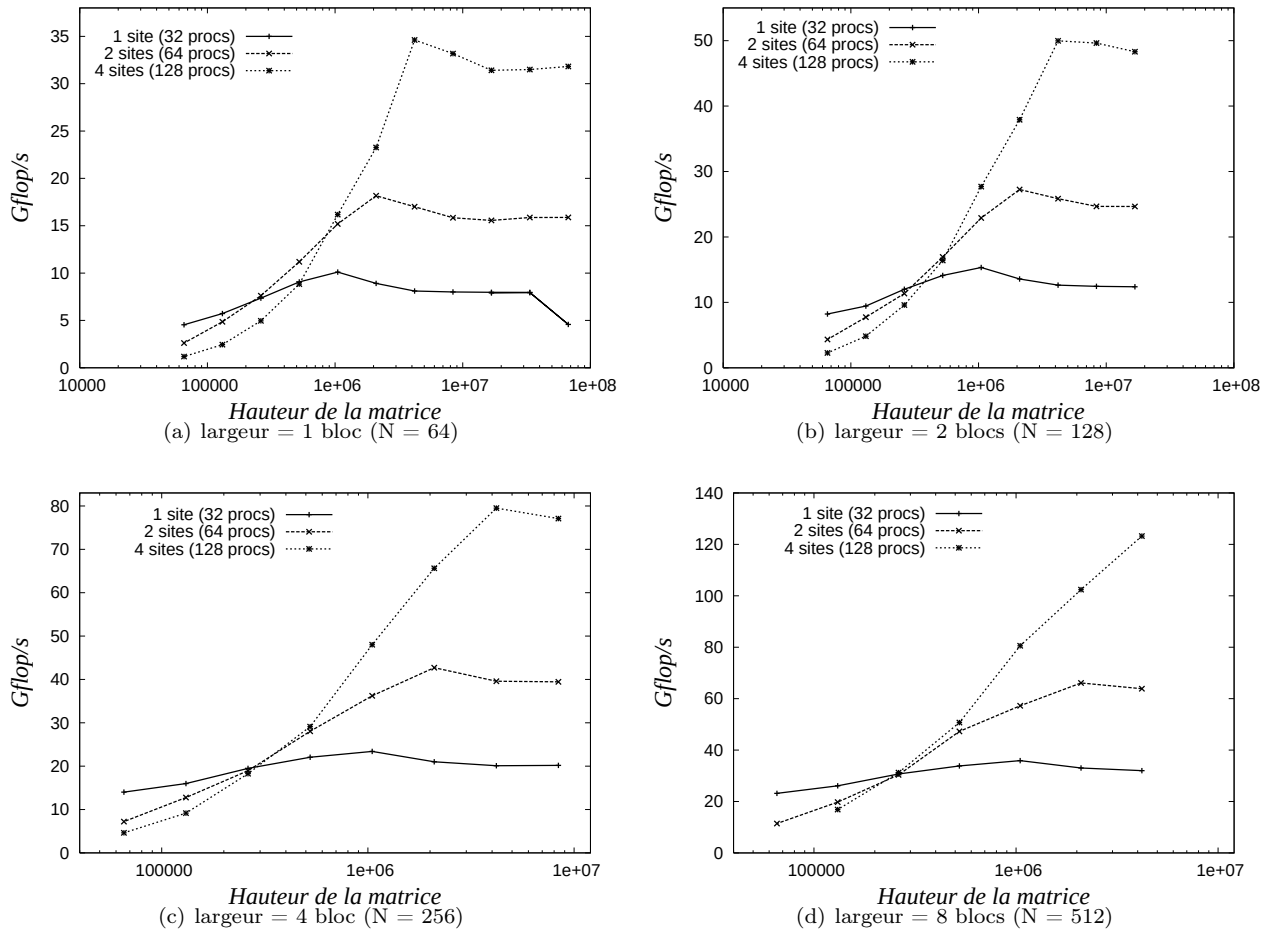
En revanche, nous nous attendons à ce qu'il importe de découper la matrice en (au moins) autant de domaines que de sites géographiques distincts. En effet, les communications inter-sites étant plus coûteuses, nous nous attendons à ce qu'augmente le seuil de hauteur de matrice à partir duquel le temps de communication d'une factorisation de type ScaLAPACK devient faible devant le temps passé en mode calcul.

Passage à l'échelle de la grille

La figure 6.19 présente une étude du passage à l'échelle de la factorisation TSQR sur une grille (Grid'5000). Nous avons utilisé 32 machines par grappe, en utilisant un processus par machine. Les mesures présentées sur la figure 6.19 ont été effectuées en configurant GotoBLAS pour utiliser un seul thread. Les quatre sous-figures correspondent à des largeurs de matrices différentes : la figure 6.19(a) représente le cas extrême, utilisant des matrices d'une largeur de un bloc de 64 réels (ce bloc correspond au découpage du domaine effectué par ScaLAPACK).

La performance augmente avec la taille de la matrice jusqu'à obtenir un pic. On constate que pour des petites matrices, on obtient de meilleures performances sur une seule grappe de 32 processus. Des petites matrices demandent beaucoup de communications par rapport à la quantité de calcul local : les communications, en particulier sur la grille, ont alors un coût important. La chute en fin de la courbe pour la mesure sur un site de la figure 6.19(a) correspond à une taille de matrice ne tenant plus en mémoire et nécessitant d'utiliser la partition d'échange sur le disque dur.

Une fois le pic atteint, on constate un rapport 1-2-4 entre l'utilisation d'un, deux et quatre sites. Sur la figure 6.19(d), la courbe de performance sur quatre sites n'atteint pas le plateau constaté une fois le pic atteint. La matrice, dont la largeur était de huit blocs (N = 512) ne tient pas en mémoire au-delà des tailles utilisées. Cependant, comme le dernier point mesuré correspond à une performance égale au double de la performance



(a) largeur = 1 bloc (N = 64)
(b) largeur = 2 blocs (N = 128)
(c) largeur = 4 bloc (N = 256)
(d) largeur = 8 blocs (N = 512)

FIGURE *6.19 – Passage à l'échelle de l'algorithme de factorisation Scalapack QR à évitement de communications sur une matrice haute et fine sur une grille utilisant 1 à 4 grappes pour différentes largeurs de la matrice et en faisant varier la hauteur.*

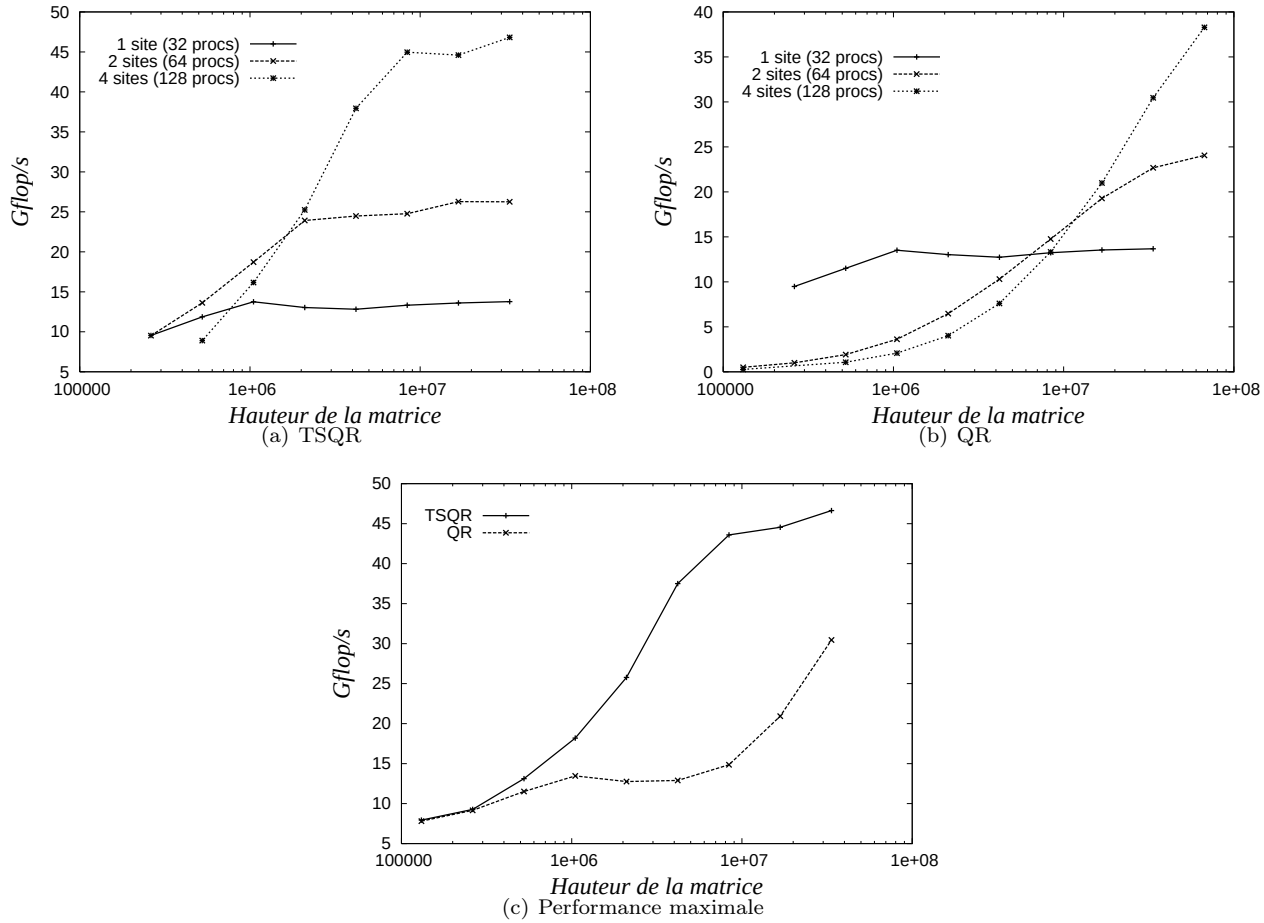
de pic observée avec deux sites, on peut penser que ce point correspond au pic.

La figure 6.20 montre le passage à l'échelle de TSQR et QR, en utilisant deux threads pour les opérations locales effectuées par GotoBLAS. On observe une meilleure performance globale : on observe sur la figure 6.20(a) de meilleures performances sur des matrices de un bloc de large que sur la figure 6.19(a), où un seul thread est utilisé. On constate la même performance entre TSQR et QR sur un seul site.

Pour les deux algorithmes de factorisation, les performances sur des petites matrices sont meilleures sur un site en raison du rapport entre les communications et les calculs évoqué ci-avant, puis les courbes se croisent et les performances sont meilleures sur des grandes machines en utilisant les quatre sites, et donc un nombre maximal de processeurs (128 processeurs).

On constate sur la figure 6.20 que les croisements ont lieu pour des matrices de plus petite taille avec TSQR. TSQR permet de limiter les communications entre les grappes (les communications les plus coûteuses), et améliore le rapport entre les communications et les calculs en faveur des calculs plus rapidement.

On peut voir sur la figure 6.20(b) qu'une exécution sur deux sites obtient la performance maximale entre les trois configurations (un, deux et quatre sites) sur un intervalle de hauteurs de matrices réduit. On peut l'expliquer par l'aspect synchrone de l'implémentation de QR de ScaLAPACK : le coût d'une communication entre des processus situés sur deux grappes étant répercuté sur tout le calcul, la différence de surcoût entre des communications sur deux grappes et sur trois grappes n'est pas très importante. C'est pourquoi les exécutions sur quatre grappes obtiennent rapidement de meilleures performances que celles sur deux grappes.



(a) TSQR (b) QR

(c) Performance maximale

FIGURE *6.20 – Passage à l'échelle de l'algorithme de factorisation Scalapack QR à évitement de communications sur une matrice haute et fine sur une grille utilisant 1 à 4 grappes pour des matrices de un bloc en largeur et en faisant varier la hauteur, en utilisant deux threads par machine.*

On peut comparer la performance maximale obtenue pour les exécutions sur un, deux et quatre sites de Scalapack QR et TSQR. La figure 6.20(c) présente cette comparaison pour des matrices de un bloc de largeur, en utilisant deux threads par machine.

Pour des matrices de petite taille, les deux algorithmes ont des performances similaires. On peut s'attendre à ce que les performances de QR converge asymptotiquement vers les performance de TSQR par valeurs inférieures, le coût des communications devenant négligeables devant celui des opérations locales. Cependant, il faudrait alors des tailles de matrices suffisantes pour que les sous-matrices sur lesquelles s'effectuent le calcul local soient assez grandes pour nécessiter un calcul d'une durée très grande devant la durée d'une phase de communications : de telles matrices ne tiennent pas en mémoire sur les machines utilisées ici.

On constate que la performance de TSQR croît plus rapidement que celle de QR. La courbe de performances de TSQR est convexe, celle de QR est concave. Pour des matrices de taille intermédiaire la factorisation TSQR obtient des performances significativement meilleures. Quel que soit le nombre de sites utilisé, TSQR obtient toujours de meilleures performances.

6.5 Conclusion

On a vu dans ce chapitre une méthode pour programmer des applications parallèles selon la norme MPI de façon efficace sur les grilles. Cette approche se base sur les possibilités fournies par le système QosCosGrid vu

au chapitre précédent en mettant à contribution le meta-ordonnanceur de grille et l'environnement d'exécution de la bibliothèque MPI.

L'approche présentée ici consiste à concevoir une application en suivant des schémas de communications et de calculs hiérarchique et en les décrivant auprès du méta-ordonnanceur de grille. Celui-ci a alors la charge d'allouer des ressources selon une topologie physique compatible avec les conditions demandées par le programmeur de l'application et nécessaire à l'exécution efficace de l'application. Le programmeur dipose alors d'une connaissance de la topologie de l'application au moment où il la conçoit.

Ce chapitre présente tout d'abord un ensemble d'algorithmes de communications collectives optimisées pour les grilles. Les expériences montrent le bénéfice de la hiérarchisation des communications afin de limiter le nombre de communications coûteuses (typiquement, entre les grappes sur une grille, ou entre les machines sur une grappe de multi-processeurs) pour la plupart des opérations collectives. Cependant, celles basées sur un algorithme mettant en place un pipeline comme une chaîne ou un anneau double sont moins performantes lorsqu'elles sont hiérarchisées.

La deuxième partie de ce chapitre présente des applications typiques programmées suivant ce principe de hiérarchisation statique. Dans un premier temps, une application constituée de phases de communications collectives et d'un calcul selon un schéma maître-esclave est hiérarchisée selon plusieurs niveaux. L'analyse des performances montre deux niveaux d'adaptation à la topologie de la grille : l'utilisation d'opérations collectives optimisées est une adaptation peu invasive tout en permettant d'améliorer les performances. Un gain plus important, notamment pour passer à l'échelle, est obtenu en adaptant les schémas de communications de l'application.

Dans un deuxième temps, une application d'algèbre linéaire est présentée. Cette famille d'applications est typiquement inadaptée à la grille du fait de ses communications synchrones et fréquentes. Les algorithmes à évitement de communications sont une nouvelle famille d'algorithmes pouvant s'adapter à la grille dans cette approche. Un algorithme de factorisation de matrices hautes et fines est examiné ici et montre de bonnes performances en passant à l'échelle et en montrant de meilleures performances que les algorithmes traditionnels sur une grille composée de plusieurs grappes.

Cette approche est particulièrement statique. Des travaux futurs pourront se poser la question des applications dynamiques, dans leurs schémas de communications et dans les processus qui les composent (création dynamique de processus, déplacement d'agents) et des topologies qui ne peuvent pas être connues à l'avance.

Les conditions exprimées auprès du système d'ordonnancement sont également statiques. En regard de la durée de vie d'une application, on peut se demander si les conditions sur les ressources demandées par le programmeur demeurent pertinentes après plusieurs années.

Chapitre 7

Conclusion

La course à la puissance de calcul passe aujourd'hui inévitablement par l'augmentation de la taille des systèmes. La fréquence des micro-processeurs étant aujourd'hui arrivée à un niveau au-delà duquel elles n'augmentent que peu, obtenir une puissance de calcul supérieure doit se faire obligatoirement par l'ajout de ressources et l'exploitation du parallélisme.

7.1 L'environnement d'exécution à grande échelle

Les systèmes de grande taille introduisent des problématiques liées au passage à l'échelle des applications parallèles exécutées, mais aussi de l'environnement qui les supporte. En particulier, l'environnement d'exécution est chargé de déployer les processus d'une application parallèle sur les ressources de calcul qui vont être utilisées, de leur permettre de communiquer les uns avec les autres, de surveiller leur état, de transmettre les signaux et les entrées-sorties et de terminer l'application à la fin de son exécution. Dans le paradigme de communication par passage de messages, en particulier par la norme MPI, une abstraction est fournie à l'application en nommant les processus de l'application par un rang. Ce nommage permet aux processus de communiquer entre eux tout en permettant la portabilité d'un système à un autre. La traduction entre les informations de contact des processus, propres à la machine sur lesquels ils s'exécutent, et ce rang générique fait également partie des rôles de l'environnement d'exécution.

7.1.1 Passage à l'échelle

Nous avons vu que l'utilisation d'un grand nombre de nœuds de calcul demande à l'environnement d'exécution d'être capable de passer à l'échelle dans ses fonctionnalités et l'infrastructure qu'il utilise pour fournir celles-ci. Le lancement d'un côté et les communications d'un autre nécessitent d'utiliser des topologies capables de supporter un grand nombre de nœuds efficacement. L'infrastructure de communications doit notamment utiliser une topologie capable de router de manière efficace les messages de signalisation de l'application (fonctionnement interne de l'environnement d'exécution et transmission des signaux et des entrées/sorties entre les processus et le mpiexec). Par ailleurs, la construction de cette infrastructure de communications doit être effectuée de façon efficace lors du lancement de l'application.

Le chapitre "Scalabilité des environnements d'exécution" montre qu'en utilisant la topologie de routage pour le déploiement de l'environnement d'exécution et en y intégrant la construction de l'infrastructure de communications on peut obtenir de bonnes performances et éviter les problèmes de congestion au niveau du processus lanceur. On a vu que l'utilisation de cette topologie seule donne des performances acceptables en communications, proches des performances obtenues avec un algorithme optimal pour les communications collectives (barrière et concaténation avec redistribution du résultat) tout en présentant l'avantage d'une topologie simple et rapide à construire.

Une topologie en arbre permet de pipeliner les étapes du lancement d'une application en ne nécessitant pas d'attendre que tous les processus impliqués aient terminé une étape pour passer à la suivante. En supprimant

ces points de synchronisation entre tous les processus on peut introduire un asynchronisme dans le lancement qui permet de pipeliner ces étapes le long de l'arbre et de raccourcir leur étalement temportel.

7.1.2 Tolérance aux pannes

La probabilité qu'un composant d'un système soit victime d'une panne augmente avec le nombre de composants le constituant. Par conséquent, il est nécessaire de pouvoir continuer une exécution malgré l'éventualité de pannes dans le système. Parmi les rôles de l'environnement d'exécution vis-à-vis de l'application s'ajoute alors le fait de pouvoir supporter un mécanisme de tolérance aux pannes au niveau applicatif. La première fonctionnalité est disponible sur les environnements d'exécution de la majorité des bibliothèques implémentant la norme MPI et est caractérisée par celle-ci : l'application doit être finalisée en cas de panne si aucun mécanisme de tolérance aux pannes n'est disponible. On peut aller au-delà de ce support minimal et proposer des mécanismes permettant à l'application de poursuivre son exécution.

Le chapitre "Tolérance aux pannes" montre d'abord le cas d'un protocole de retour sur points de reprise coordonné où le rôle de l'environnement d'exécution est réduit à l'enregistrement des points de reprise des processus et à une adaptation du mécanisme de relancement. On y voit cependant que cette famille de protocoles ne passe pas à l'échelle : on doit alors s'orienter vers des protocoles où l'environnement d'exécution joue un rôle plus important et prend une part active dans la tolérance aux pannes avec les protocoles non-coordonnés.

En effet, si avec les protocoles coordonnés l'environnement d'exécution peut se terminer et être relancé lorsqu'une panne survient, dans le cas des protocoles non-coordonnés il doit survivre à la panne et continuer de fournir ses services, éventuellement en mode dégradé, aux processus survivants. Ce chapitre présente un mécanisme permettant à l'environnement d'exécution de rétablir son état tout en poursuivant son exécution auprès des processus qui n'ont pas été touchés par la panne. L'évaluation de performances a montré qu'outre le fait de laisser les processus qui le peuvent poursuivre leur exécution, l'intégration d'un nouveau nœud dans l'environnement d'exécution et le relancement et l'intégration dans l'application des processus qui ont été victimes de la panne prend un temps moins important que le lancement de l'application entière.

7.1.3 Cas des grilles de calcul

Il est possible de former un système à grande échelle en agrégeant des systèmes de plus petite taille, parfois géographiques éloignés les uns des autres. Cependant, les politiques de sécurité indispensables au maintien de l'intégrité des grappes impliquées (pare-feux, traductions d'adresse) sont des obstacles à la connectivité entre processus exécutés sur différentes grappes.

Le chapitre "Intergiciel pour MPI sur les grilles" présente l'architecture de la pile logicielle QosCosGrid permettant d'exploiter ces grilles de calcul. En particulier, l'attention se porte sur QCG-OMPI, l'intergiciel de communications permettant d'assurer la connectivité entre les processus de l'application.

En se basant sur une extension de l'environnement d'exécution d'une bibliothèque MPI existante, cet intergiciel fournit à l'application une interface de communication transparente suivant la norme MPI permettant d'utiliser des techniques d'interconnexion avancées de façon transparente. Ainsi deux processus situés de part et d'autre d'un équipement de sécurité ont la possibilité de communiquer ensemble sans se rendre compte qu'ils passent d'une grappe à une autre ni qu'ils sont en train de passer à travers un pare-feu.

QCG-OMPI rend donc possible l'utilisation d'une grille semblable du point de vue de l'application à une grappe. Or, les performances des communications au sein d'une grille diffèrent de plusieurs ordres de grandeur selon le niveau de hiérarchie où elles ont lieu : entre deux cœurs d'une même machine, entre deux machines d'une même grappe ou entre deux grappes. Les applications doivent donc utiliser des schémas de communications adaptés à cette hiérarchie de performances.

Le chapitre "Applications MPI pour les grilles" présente une nouvelle méthode pour programmer des applications pour les grilles de calcul. Puisqu'il est difficile de programmer une application s'adaptant de manière efficace à n'importe quelle topologie physique, le système QosCosGrid permet de mettre à contribution son méta-ordonnanceur de grille afin d'obtenir une topologie physique correspondant à celle pour laquelle l'application à été conçue. Ainsi, l'application doit être adaptée à une topologie fixe, connue du programmeur.

Ce chapitre présente tout d'abord un ensemble d'opérations collectives optimisées permettant un premier niveau d'adaptation topologique des applications. Ensuite une méthode de programmation des grilles de calcul

est présentée afin de tirer parti des performances d'une topologie logique correspondant au moins à une topologie logique paramétrée et fournie au méta-ordonnanceur de grille.

Une fois l'application lancée, l'environnement d'exécution dispose d'informations sur cette topologie virtuelle permettant à l'application de s'organiser en retrouvant sa structure. Ces informations sont alors passées aux processus de l'application en utilisant des appels à des extensions de la norme MPI.

Deux applications sont présentées pour montrer l'efficacité de cette méthode. La première est une application de géophysique utilisant des opérations collectives et un calcul suivant le schéma maître-esclave. Il est montré l'intérêt de l'utilisation de schémas hiérarchiques pour les opérations collectives puis pour le calcul en utilisant un schéma maître-esclave hiérarchique. La seconde application présentée ici est une routine de base de résolution d'un problème d'algèbre linéaire effectuant la triangularisation d'une matrice. Cette application démontre une adaptativité particulièrement bonne pour les grilles de calcul en exploitant les schémas hiérarchiques d'une famille d'algorithmes dits à évitement de communications.

7.2 Perpectives

Les travaux menés au cours de cette thèse ouvrent de nouvelles questions sur les environnements d'exécution. Certaines sont déjà en cours d'investigation, d'autres demeurent pour le moment complètement ouvertes.

7.2.1 En voyant plus loin

Les travaux présentés ici ouvrent des portes sur des sujets directement liés à ce qui a été démontré ici

Intégration avec les systèmes de lancement On a vu l'importance de la finesse de l'intégration de l'environnement d'exécution avec le système de lancement, notamment pour la construction de l'infrastructure de communications. Cette intégration fine a été rendue possible en utilisant un lanceur intégré à l'environnement d'exécution utilisant directement SSH. À l'inverse, les lanceurs spécialisés utilisés par l'environnement d'exécution comme des applications tiers ne permettent pas actuellement cette intégration fine. L'absence d'interaction entre le lanceur et l'environnement d'exécution durant son déploiement et notamment la construction de l'infrastructure de communications ne permet pas de mettre en place de mécanismes efficaces de construction de cette infrastructure mais obligent à la construire de façon centralisée, ce qui provoque à grande échelle des tempêtes de connexions. Une intégration plus fine avec les systèmes de lancement et notamment une l'utilisation de leurs fonctionnalités de communications permettrait d'exploiter de façon plus performante leur lanceur en mettant en place des mécanismes efficaces de construction de l'environnement d'exécution.

Autostabilitation pour le maintien de l'environnement d'exécution L'autostabilisation [150, 71] apporte des techniques de construction et de maintien de topologies de communications ayant des propriétés requises par les infrastructures de communications des environnements d'exécution. Des résultats ont déjà été obtenus dans ce sens [23, 22] et montrent l'intérêt présenté par cette voie. L'implémentation des algorithmes auto-stabilisants dans des systèmes réels donnerait le jour à une nouvelle génération d'infrastructures de communications hautement résilientes.

Algèbre linéaire sur grille Les résultats obtenus dans cette thèse sur une application d'algèbre linéaire montrent pour la première fois qu'il est possible d'effectuer des calculs de ce type sur une grille. Il serait donc intéressant de généraliser les résultats obtenus à des matrices carrées, puis d'explorer d'autres algorithmes comme les deux autres grands algorithmes de factorisations, LU et Cholesky, pour lesquels il existe également des variantes à évitement de communications. Enfin, on aperçoit la possibilité d'ajouter un niveau de hiérarchie en combinant ces algorithmes sur des grappes de grappes de machines multi-cœurs.

Autres familles de schémas de communications L'environnement d'exécution de QCG-OMPI permet de supporter des applications dont le schéma de communications est connu à l'avance (au moment de soumettre l'exécution de l'application au méta-ordonnanceur de grille) et demeure fixe tout au long de l'exécution. On peut imaginer une solution au problème d'une topologie de communication changeant au cours de l'exécution

mais connue à l'avance en procédant à une migration des processus de l'application par point de reprise et modification de l'ordonnancement des ressources. Cependant, la question de l'adaptation des ressources de calcul à une application dont les schémas de communications sont inconnus et potentiellement changeants au cours de l'exécution demeure ouverte.

7.2.2 Orientations à long terme

Les enseignements apportés par les résultats de cette thèse apportent des indications sur des perspectives et des approches à suivre à plus long terme.

Modularité Les rôles de base d'un environnement d'exécution sont bien définis, mais on a vu qu'il est possible de lui demander plus afin de supporter une fonctionnalité particulière, comme la tolérance aux pannes. La clé de l'extensibilité d'un intergiciel est dans la modularité et la possibilité d'ajouter de nouvelles fonctionnalités ou de les mettre en place d'une façon différente.

Unification des environnements d'exécution On a vu dans l'introduction que les environnements d'exécution avaient une tendance cyclique à s'unifier autour d'un environnement commun à plusieurs intergiciels de communications puis à se séparer avec un environnement par intergiciel. La tendance actuelle et l'observation des projets actuels et en cours de formation laisse à penser que l'on est actuellement en train de se diriger vers une réunification de l'environnement d'exécution, avec un intergiciel de nouvelle génération comme STCI [35]. Capable d'utiliser des réseaux à hautes performances et non plus limité à TCP, on peut imaginer qu'il sera possible de l'étendre pour les langages parallèles à espace d'adressage partitionné.

Tolérance aux pannes dirigée par l'application FT-MPI [74] et la bibliothèque d'algèbre linéaire tolérant aux pannes ABFT [51] ont apporté les premières expérimentations en matière d'applications contrôlant la tolérance aux pannes. Cette approche, non transparente et permettant une tolérance aux pannes plus fine que l'approche transparente, peut tirer parti d'un environnement d'exécution tolérant aux pannes tel que celui décrit dans cette thèse. La prochaine version majeure de la norme MPI (MPI3) devrait fixer un standard pour une interface permettant à l'application de diriger la tolérance aux pannes dont elle bénéficie et d'avoir le contrôle de sa restauration d'état.

Exploitation de la hiérarchie dans les machines à grande échelle Les machines à très grande échelle sont construites selon un schéma hiérarchique, notamment au niveau du réseau. En effet, il n'est pas possible d'interconnecter de façon uniforme tous les nœuds d'une machine à très grande échelle. On est alors obligé de mettre en place une hiérarchie dans les commutateurs. De plus, on s'oriente résolument vers des machines constituées de nœuds comportant de nombreux cœurs (plusieurs centaines). Il va donc être nécessaire de procéder pour ces machines aussi à une hiérarchisation des communications collectives dans un premier temps, puis, s'il s'avère que les différents niveaux de hiérarchie présentent des différences de performances trop importantes, des schémas de calcul.

Bibliographie

[1] Marcos Kawazoe Aguilera, Wei Chen and Sam Toueg. « Heartbeat : A Timeout-Free Failure Detector for Quiescent Reliable Communication ». In Marios Mavronicolas and Philippas Tsigas, editors, *Proceedings of the 11th Workshop on Distributed Algorithms (WDAG'97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 1997.

[2] Roberto Alfieri, Roberto Cecchini, Vincenzo Ciaschini, Luca dell'Agnello, Ákos Frohner, Alberto Gianoli, Károly Lörentey and Fabio Spataro. « VOMS, an Authorization System for Virtual Organizations ». In F. Fernández Rivera, Marian Bubak, A. Gómez Tato and Ramon Doallo, editors, *European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 33–40. Springer, 2003.

[3] George Almási, Charles Archer, José G. Castaños, Manish Gupta, Xavier Martorell, José E. Moreira, William D. Gropp, Silvius Rus and Brian R. Toonen. « MPI on BlueGene/L : Designing an Efficient General Purpose Messaging Solution for a Large Cellular System ». 2840 :352–361, 2003.

[4] M. Nedim Alpdemir, Arijit Mukherjee, Anastasios Gounaris, Alvaro A. A. Fernandes, Norman W. Paton and Paul Watson. « An Experience Report on Designing and Building OGSA-DQP : A Service Based Distributed Query Processor for the Grid ». October 2003.

[5] David P. Anderson. « BOINC : A System for Public-Resource Computing and Storage ». In Rajkumar Buyya, editor, *Proceedings of the 5th International Workshop on Grid Computing (GRID'04)*, pages 4–10, Pittsburgh, PA, USA, November 2004. IEEE Computer Society.

[6] Thara Angskun, George Bosilca and Jack Dongarra. « Binomial Graph : A Scalable and Fault-Tolerant Logical Network Topology ». In Ivan Stojmenovic, Ruppa K. Thulasiram, Laurence Tianruo Yang, Weijia Jia, Minyi Guo and Rodrigo Fernandes de Mello, editors, *Proceedings of the 5th International Symposium on Parallel and Distributed Processing and Applications (ISPA 2007)*, volume 4742 of *Lecture Notes in Computer Science*, pages 471–482. Springer, 2007.

[7] Thara Angskun, Graham Fagg, George Bosilca, Jelena Pjesivac-Grbovic and Jack Dongarra. « Scalable fault tolerant protocol for parallel runtime environments ». In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'06)*, pages 141–149. Springer, 2006.

[8] Lario Antonioletti, Neil Chue Hong, Ally C. Hume., Mike Jackson, Amy Krause, Jeremy Nowell, Charaka Palansuriya., Tom Sugden and MArtin Westhead. « Experiences of Designing and Implementing Grid Database Services in the OGSA-DAI Project ». In *Designing and Building Grid Services Workshop, GGF9*, October 2003.

[9] Dorian C. Arnold and Barton P. Miller. « A Scalable Failure Recovery Model for Tree-based Overlay Networks ». Technical Report TR1626, University of Wisconsin, Computer Science Department.

[10] Dorian C. Arnold, G. D. Pack and Barton P. Miller. « Tree-based overlay networks for scalable applications ». In *Proceedings of the 21st International Parallel & Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[11] Emanouil I. Atanassov, Todor V. Gurov, Aneta Karaivanova and Mihail Nedjalkov. « Monte Carlo Grid Application for Electron Transport ». In Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot and Jack Dongarra, editors, *Proceedings of the 6th International Conference on Computational*

Science (ICCS'06), Part III, volume 3993 of *Lecture Notes in Computer Science*, pages 616–623. Springer, 2006.

[12] Olivier Aumage, Luc Bougé, Jean-François Méhaut and Raymond Namyst. « Madeleine II : A Portable and Efficient Communication Library for High-Performance Cluster Computing ». *Parallel Computing*, 28(4) :607–626, April 2002.

[13] Algirdas Avizienis, Jean-Claude Laprie and Brian Randell. « Dependability and its threats - A taxonomy ». In René Jacquart, editor, *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, pages 91–120. Kluwer, 2004.

[14] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell and Carl E. Landwehr. « Basic Concepts and Taxonomy of Dependable and Secure Computing ». *IEEE Trans. Dependable Sec. Comput.*, 1(1) :11–33, 2004.

[15] Rosa M. Badia, D. Du, Eduardo Huedo, A. Kokossis, Ignacio Martín Llorente, Rubén S. Montero, Marc de Palol, Raúl Sirvent and Constantino Vázquez. « Integration of GRID Superscalar and GridWay Metascheduler with the DRMAA OGF Standard ». In Emilio Luque, Tomàs Margalef and Domingo Benitez, editors, *Proceedings of the 14th European Conference on Parallel and Distributed Computing (EuroPar'08)*, volume 5168 of *Lecture Notes in Computer Science*, pages 445–455. Springer, 2008.

[16] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo and Maurice Yarrow. « The NAS Parallel Benchmarks 2.0 ». Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, 1995.

[17] Pavel Bar, Camille Coti, Derek Groen, Thomas Herault, Valentin Kravtsov, Assaf Schuster and Martin Swain. « Running parallel applications with topology-aware grid middleware ». In Paul Roe and David Wallom, editors, *5th IEEE International Conference on e-Science (eScience 2009)*, December 2009. to appear.

[18] Micah Beck, Jack Dongarra and James S. Plank. « NetSolve/D : A Massively Parallel Grid Execution System for Scalable Data Intensive Collaboration ». In *Proceedings of the 19th International Parallel and Distributed Processing Symposium CD-ROM (19th IPDPS'05)*, Denver, CO, USA, April 2005. IEEE Computer Society (Los Alamitos, CA).

[19] Dan Boachea. « GASNet Specification, v1.1 ». Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.

[20] Dan Bonachea and Jason Duell. « Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations ». *International Journal of High Performance Computing and Networking (IJHPCN)*, 1(1/2/3) :91–99, 2004.

[21] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fédak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri and Anton Selikhov. « MPICH-V : Toward a Scalable Fault Tolerant MPI for Volatile Nodes ». In *High Performance Networking and Computing (SC2002)*, Baltimore USA, November 2002. IEEE/ACM.

[22] George Bosilca, Camille Coti, Thomas Herault, Pierre Lemarinier and Jack Dongarra. « Constructing Resiliant Communication Infrastructure for Runtime Environments ». In *International Conference in Parallel Computing (ParCo2009)*, Lyon, France, September 2009.

[23] George Bosilca, Camille Coti, Thomas Herault, Pierre Lemarinier and Jack Dongarra. « Constructing Resiliant Communication Infrastructure for Runtime Environments ». Technical Report ICL-UT-09-02, Innovative Computing laboratory, University of Tennessee, http ://icl.eecs.utk.edu/publications/, 2009.

[24] George Bosilca, Remi Delmas, Jack Dongarra and Julien Langou. « Algorithm-based fault tolerance applied to high performance computing ». *J. Parallel Distrib. Comput.*, 69(4) :410–416, 2009.

[25] Fatiha Bouabache, Thomas Hérault, Gilles Fedak and Franck Cappello. « Hierarchical Replication Techniques to Ensure Checkpoint Storage Reliability in Grid Environment ». In *Proceedings of the 8th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'08)*, pages 475–483, 2008.

[26] Salah-Salim Boutammine, Daniel Millot and Christian Parrot. « An Adaptive Scheduling Method for Grid Computing ». In Wolfgang E. Nagel, Wolfgang V. Walter and Wolfgang Lehner, editors, *Proceedings of the 12th European Conference on Parallel and Distributed Computing (EuroPar'06)*, volume 4128 of *Lecture Notes in Computer Science (LNCS)*, pages 188–197, Dresden, Germany, August-September 2006. Springer-Verlag (Berlin/New York).

[27] Aurélien Bouteiller. « *Tolérance automatique aux défaillances par points de reprise et retour en arrière dans les systèmes hautes performances à passage de messages* ». Doctorat en sciences, spécialité informatique, Université Paris-Sud XI - Orsay, 2006.

[28] Aurélien Bouteiller, George Bosilca and Jack Dongarra. « Redesigning the Message Logging Model for High Performance ». In *International Supercomputer Conference (ISC 2008), Dresden, Germany*, June 2008.

[29] Aurélien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier and Frédéric Magniette. « MPICH-V2 : a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging ». In *High Performance Networking and Computing (SC2003)*. Phoenix USA, IEEE/ACM, November 2003.

[30] Aurelien Bouteiller, Boris Collin, Thomas Herault, Pierre Lemarinier and Franck Cappello. « Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI ». In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 97, Washington, DC, USA, 2005. IEEE Computer Society.

[31] Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik and Franck Cappello. « Coordinated checkpoint versus message log for fault tolerant MPI ». In *IEEE International Conference on Cluster Computing (Cluster 2003)*. IEEE CS Press, December 2003.

[32] Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik and Franck Cappello. « Coordinated checkpoint versus message log for fault tolerant MPI ». *International Journal of High Performance Computing and Networking (IJHPCN)*, (3), 2004.

[33] Simon Branford, Cihan Sahin, Ashish Thandavan, Christian Weihrauch, Vassil N. Alexandrov and Ivan TomovDimov. « Monte Carlo methods for matrix computations on the grid ». *Future Gener. Comput. Syst.*, 24(6) :605–612, 2008.

[34] Joshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal and Derrick Weathersby. « Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems ». *IEEE Transactions on Parallel and Distributed Systems*, 8(11) :1143–1156, November 1997.

[35] Darius Buntinas, George Bosilca, Richard L. Graham, Geoffroy Vallée and Gregory R. Watson. « A Scalable Tools Communications Infrastructure ». In *Proceedings of the High Performance Computing Symposium (HPCS'08)*, pages 33–39, 2008.

[36] Darius Buntinas, Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez and Franck Cappello. « Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI ». *Future Generation Computer Systems*, 24 (1) :73–84, 2008. Digital Object Identifier : http ://dx.doi.org/10.1016/j.future.2007.02.002.

[37] Darius Buntinas, Guillaume Mercier and William D. Gropp. « Design and Evaluation of Nemesis : a Scalable, Low-Latency, Message-Passing Communication Subsystem ». In *Proceedings of the 6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'06)*, Singapore, May 2006. Held in conjunction with IEEE Computer Society and ACM.

[38] Greg Burns, Raja Daoud and James Vaigl. « LAM : An Open Cluster Environment for MPI ». In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[39] Ralph M. Butler, William D. Gropp and Ewing L. Lusk. « A Scalable Process-Management Environment for Parallel Programs ». In Jack Dongarra, Péter Kacsuk and Norbert Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'02)*, volume 1908, pages 168–175. Springer, 2000.

[40] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron and Olivier Richard. « A batch scheduler with high level components ». In

Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGRID'05), pages 776–783, Cardiff, UK, May 2005. IEEE Computer Society.

[41] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Yvon Jegou, Pascale Vicat-Blanc Primet, Emmanuel Jeannot, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Benjamin Quetier and Olivier Richard. « Grid'5000 : A Large Scale and Highly Reconfigurable Grid Experimental Testbed ». In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing CD (SC|05)*, pages 99–106, Seattle, Washington, USA, November 2005. IEEE/ACM.

[42] Franck Cappello, Pierre Fraigniaud, Bernard Mans and Arnold L. Rosenberg. « HiHCoHP : Toward a Realistic Communication Model for Hierarchical HyperClusters of Heterogeneous Processors ». In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS'01)*, page 42, San Francisco, CA, USA, April 2001. IEEE Computer Society (Los Alamitos, CA).

[43] David Carmeli, Valentin Kravtsov, Benny Yoshpa, Assaf Schuster, Krzysztof Kurowski, Camille Coti and Thomas Herault. « D2.1 Part 1 : Grid Services for Quasi Opportunistic Super Computing ». Technical Report, European Union, QosCosGrid project FP6 IST-2005-033883, April 2007.

[44] Denis Caromel, Christian Delbe, Alexandre Di Costanzo and Mario Leyton. « ProActive : an Integrated platform for programming and running applications on grids and P2P systems », 2006.

[45] Henri Casanova and Jack Dongarra. « NetSolve : A Network-enabled Server for Solving Computational Science Problems ». *The International Journal of Supercomputer Applications and High Performance Computing*, 11 :212–223, 2000.

[46] Ralph H. Castain, Timothy S. Woodall, David J. Daniel, Jeffrey M. Squyres, Brian Barrett and Graham E. Fagg. « The Open Run-Time Environment (OpenRTE) : A Transparent Multi-cluster Environment for High-Performance Computing ». In Beniamino Di Martino, Dieter Kranzlmüller and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 225–232, Sorrento, Italy, September 2005. Springer.

[47] Marco Cecchi, Fabio Capannini, Alvise Dorigo, Antonia Ghiselli, Francesco Giacomini, Alessandro Maraschini, Moreno Marzolla, Salvatore Monforte, Fabrizio Pacini, Luca Petronzio and Francesco Prelz. « The gLite Workload Management System ». In Nabil Abdennadher and Dana Petcu, editors, *Proceedings of the 4th Onternational Conference on Advances in Grid and Pervasive Computing (GPC'09)*, volume 5529 of *Lecture Notes in Computer Science*, pages 256–268, Geneva, Switzerland, May 2009. Springer.

[48] Christophe Cérin, Jean-Christophe Dubacq and Jean-Louis Roch. « Methods for Partitioning Data to Improve Parallel Execution Time for Sorting on Heterogeneous Clusters ». In Yeh-Ching Chung and José E. Moreira, editors, *GPC*, volume 3947 of *Lecture Notes in Computer Science*, pages 175–186. Springer, 2006.

[49] K. Mani Chandy and Leslie Lamport. « Distributed Snapshots : Determining Global States of Distributed Systems ». In *Transactions on Computer Systems*, volume 3(1), pages 63–75. ACM, February 1985.

[50] Qun Chen and Michael C. Ferris. « FATCOP : A Fault Tolerant Condor-PVM Mixed Integer Program Solver ». *SIAM Journal on Optimization*, 4, 2001.

[51] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca and Jack Dongarra. « Fault tolerant high performance computing by a coding approach ». In Keshav Pingali, Katherine A. Yelick and Andrew S. Grimshaw, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, pages 213–223. ACM, 2005.

[52] Jaeyoung Choi, James Demmel, Inderjit S. Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David W. Walker and R. Clinton Whaley. « ScaLAPACK : A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance ». In *PARA*, pages 95–106, 1995.

[53] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker and R. Clinton Whaley. « A Proposal for a Set of Parallel Basic Linear Algebra Subprograms ». In *PARA*, pages 107–114, 1995.

[54] Benoit Claudel, Guillaume Huard and Olivier Richard. « TakTuk, adaptive deployment of remote executions ». In Dieter Kranzlmüller, Arndt Bode, Heinz-Gerd Hegering, Henri Casanova and Michael Gerndt, editors, *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, (HPDC'09)*, pages 91–100, Garching, Germany, 2009. ACM.

[55] C. Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, Daniel Chavarria-Miranda, Francois Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti and Yiyi Yao. « An Evaluation of Global Address Space Languages : Co-Array Fortran and Unified Parallel ». June 2005.

[56] Denis Conan. « *Tolérance aux fautes par recouvrement arrière dans les systèmes informatiques répartis* ». Doctorat en sciences, spécialité informatique, Université Paris 6, September 1996.

[57] Mike Cooke. « Silicon transistor hits 500GHz performance ». *III-Vs Review*, 19(5) :30 – 31, 2006.

[58] Camille Coti, Thomas Herault and Franck Cappello. « MPI Applications on Grid : A Topology-Aware approach ». Technical Report 6633, INRIA, September 2008.

[59] Camille Coti, Thomas Herault and Franck Cappello. « MPI Applications on Grids : a Topology-Aware Approach ». In Henk Sip and Dick Epema, editors, *Proceedings of the 15th European Conference on Parallel and Distributed Computing (EuroPar'09)*, pages 466–477, Delft, the Netherlands, August 2009. LNCS.

[60] Camille Coti, Thomas Herault, Derek Groen and Mariusz Mamonski. « D1.2c : Adapted Version of the OpenMPI Communication Library ». Technical Report, European Union, QosCosGrid project FP6 IST-2005-033883, May 2009.

[61] Camille Coti, Thomas Herault, Pierre Lemarinier, Sylvain Peyronnet and Ala Rezmerita. « D1.2a : OpenMPI Communication Library ». Technical Report, European Union, QosCosGrid project FP6 IST-2005-033883, October 2007.

[62] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez and Franck Cappello. « Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI ». In ACM/IEEE, editor, *Proceedings of the International Conference for High Performance Networking Computing, Networking, Storage and Analysis (SC|06)*, page electronic, Tampa, USA, November 2006.

[63] Camille Coti, Thomas Herault, Sylvain Peyronnet, Ala Rezmerita and Franck Cappello. « Grid Services For MPI ». In Thierry Priol et al, editor, *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08)*, pages 417–424, Lyon, France, May 2008. ACM/IEEE.

[64] Camille Coti, Thomas Herault and Ala Rezmerita. « D1.2b : Adapted Version of the OpenMPI Communication Library ». Technical Report, European Union, QosCosGrid project FP6 IST-2005-033883, October 2008.

[65] Camille Coti, Ala Rezmerita, Thomas Herault and Franck Cappello. « Grid Services For MPI ». In Franck Cappello, Thomas Herault and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'07)*, pages 393–394, Paris, France, October 2007. ACM/IEEE.

[66] W. Dally and C. L. Seitz. « Deadlock-Free Message Routing in Multiprocessor Interconnection Networks ». *IEEE Trans. Computs.*, C-36(5) :547–553, 1987.

[67] Vincent Danjean, Roland Gillard, Serge Guelton, Jean-Louis Roch and Thomas Roche. « Adaptive loops with KAAPI on multicore and grid : applications in symmetric cryptography ». In Marc Moreno Maza and Stephen M. Watt, editors, *Proceedings of the International Workshop on Parallel Symbolic Computing (PASCO'07)*, pages 33–42, London, Ontario, Canada, July 2007. ACM.

[68] James Demmel, Laura Grigori, Mark Hoemmen and Julien Langou. « Communication-avoiding parallel and sequential QR factorizations ». *CoRR*, abs/0806.2159, 2008.

[69] Alexandre Denis, Olivier Aumage, Rutger F. H. Hofman, Kees Verstoep, Thilo Kielmann and Henri E. Bal. « Wide-Area Communication for Grids : An Integrated Solution to Connectivity, Performance and Security Problems ». In *Proceedings of the 13th International Symposium on High-Performance (HPDC'04)*, pages 97–106. IEEE Computer Society, 2004.

[70] Alexandre Denis, Christian Pérez and Thierry Priol. « PadicoTM : an open integration framework for communication middleware and runtimes ». *Future Generation Comp. Syst.*, 19(4) :575–585, 2003.

[71] S Dolev. *Self-Stabilization*. MIT Press, 2000.

[72] Elmootazbellah Elnozahy, Lorenzo Alvisi, Yi-Min Wang and David B. Johnson. « A survey of rollback-recovery protocols in message-passing systems ». *ACM Computing Surveys (CSUR)*, 34(3) :375 – 408, september 2002.

[73] G. E. Fagg. « HARNESS : Heterogeneous Adaptable Reconfigurable Networked Systems. Final Progress Report », April 23 2007.

[74] Graham E. Fagg and Jack Dongarra. « FT-MPI : Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World », 2000.

[75] Graham E. Fagg and Jack J. Dongarra. « HARNESS fault tolerant MPI design, usage and performance issues ». *Future Generation Computer Systems*, 18(8) :1127–1142, October 2002.

[76] Gilles Fedak, Cécile Germain, Vincent Néri and Franck Cappello. « XtremWeb : A Generic Global Computing System ». In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'01)*, pages 582–587. IEEE Computer Society, 2001.

[77] Michael J. Flynn. « Some Computer Organizations and Their Effectiveness ». *IEEE Trans. Comput.*, 21(9) :948–960, September 1972.

[78] Bryan Ford, Pyda Srisuresh and Dan Kegel. « Peer-to-Peer Communication Across Network Address Translators ». In *USENIX Annual Technical Conference, General Track (USENIX '05)*, pages 179–192. USENIX, March 18 2006.

[79] Message Passing Interface Forum. « MPI : A Message-Passing Interface Standard ». Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, April 1994. Tue, 22 May 101 17 :44 :55 GMT.

[80] I. Foster and N. Karonis. « A grid-enabled MPI : Message passing ftin heterogeneous distributed computing systems ». In *Proceedings of the International Conference on High Performance Networking and Computing (SC98)*. IEEE/ACM, November 1998.

[81] Ian Foster, Carl Kesselman and Steven Tuecke. « The Nexus Task-parallel Runtime System ». In *In Proc. 1st Intl Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994.

[82] Ian T. Foster. « What is the Grid ? A Three Point Checklist ». July 2002.

[83] Ian T. Foster. « Globus Toolkit Version 4 : Software for Service-Oriented Systems ». *J. Comput. Sci. Technol*, 21(4) :513–520, 2006.

[84] Geoffrey Fox, Sanjay Ranka, Michael L. Scott, Allen D. Malony, James C. Browne, Marina C. Chen, Alok N. Choudhary, Thomas Cheatham, Janice E. Cuny, Rudolf Eigenmann, Amr F. Fahmy, Ian T. Foster, Dennis Gannon, Tomasz Haupt, Carl Kesselman, Charles Koelbel, Wei Li, Monica S. Lam, Thomas J. LeBlanc, Jim Openshaw, David A. Padua, Constantine D. Polychronopoulos, Joel H. Saltz, Alan Sussman, Gil Weigand and Katherine A. Yelick. « Common Runtime Support for High Performance Parallel Languages ». In *Proceedings of the Supercomputing '93 Conference*, pages 752–757, 1993.

[85] 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 *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04)*, pages 97–104, Budapest, Hungary, September 2004.

[86] Edgar Gabriel, Michael Resch, Thomas Beisel and Rainer Keller. « Distributed Computing in a heterogenous computing environment ». Technical Report, 1998.

[87] Edgar Gabriel, Michael M. Resch, Thomas Beisel and Rainer Keller. « Distributed Computing in a Heterogeneous Computing Environment ». In Vassil N. Alexandrov and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 5th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'98)*, volume 1497 of *Lecture Notes in Computer Science*, pages 180–187. Springer, 1998.

[88] François Gallilée, Jean-Louis Roch, Gerson G. H. Cavalheiro and Mathias Doreille. « Athapascan-1 : On-Line Building Data Flow Graph in a Parallel Language ». In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 88–95, Paris, October 12–18, 1998. IEEE Computer Society Press.

[89] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken and P. Vranas. « Overview of the Blue Gene/L system architecture ». *IBM Journal of Research and Development*, 49(2/3) :195–212, 2005.

[90] Thierry Gautier, Xavier Besseron and Laurent Pigeon. « KAAPI : A thread scheduling runtime system for data flow computations on cluster of multi-processors ». In Marc Moreno Maza and Stephen M. Watt, editors, *Proceedings of the International Workshop on Parallel Symbolic Computing (PASCO'07)*, pages 15–23, London, Ontario, Canada, July 2007. ACM.

[91] Stéphane Genaud, Arnaud Giersch and Frédéric Vivien. « Load-balancing scatter operations for grid computing ». *Parallel Computing*, 30(8) :923–946, 2004.

[92] Stéphane Genaud and Choopan Rattanapoka. « Fault Management in P2P-MPI ». In Christophe Cérin and Kuan-Ching Li, editors, *Proceedings of Advances in Grid and Pervasive Computing, Second International Conference (GPC 2007)*, volume 4459 of *Lecture Notes in Computer Science*, pages 64–77. Springer, 2007.

[93] Yiannis Georgiou, Julien Leduc, Brice Videau, Johann Peyrard and Olivier Richard. « A tool for environment deployment in clusters and light grids ». In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, 2006.

[94] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, second edition, 1989.

[95] Kazushige Goto and Robert A. van de Geijn. « High-performance implementation of the level-3 BLAS ». *ACM Trans. Math. Softw.*, 35(1), 2008.

[96] William D. Gropp and Ewing L. Lusk. « The MPI communication library : its design and a portable implementation ». In *Proceedings of the Scalable Parallel Libraries Conference, October 6–8, 1993, Mississippi State, Mississippi*, pages 160–165, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, October 1994. IEEE Computer Society Press.

[97] William D. Gropp and Ewing L. Lusk. « MPICH working note : Creating a new MPICH device using the channel interface ». Technical Report ANL/MCS-TM-213, Argonne National Laboratory, 1995.

[98] William D. Gropp and Ewing L. Lusk. « Fault Tolerance in MPI Programs ». Technical Report ANL/MCS-P1154-0404, Argonne National Laboratory, Mathematics and Computer Science Division, April 2004.

[99] William D. Gropp, Ewing L. Lusk, Nathan Doss and Anthony Skjellum. « High-Performance, Portable Implementation of the MPI Message Passing Interface Standard ». *Parallel Computing*, 22(6) :789–828, September 1996.

[100] Marc Grunberg, Stephane Genaud and Catherine Mongenet. « Parallel Seismic Ray Tracing in a Global Earth Model ». In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, (PDPTA'02)*, volume 3, pages 1151–1157, Las Vegas, Nevada, USA, June 2002. CSREA Press.

[101] Martin Gudgin and Marc Hadley. « SOAP Version 1.2 Message Normalization ». World Wide Web Consortium, Note NOTE-soap12-n11n-20031008, October 2003.

[102] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su and K. Yelick. « Titanium Language Reference Manual ». Technical Report UCB/CSD-2005-15, U.C. Berkeley, 2005.

[103] Eduardo Huedo, Rubén S. Montero and Ignacio Martín Llorente. « The GridWay Framework for Adaptive Scheduling and Execution on Grids ». *Scalable Computing : Practice and Experience*, 6(3) :1–8, 2005.

[104] Joshua Hursey, Timothy Mattox and Andrew Lumsdaine. « Interconnect agnostic checkpoint/restart in open MPI ». In Dieter Kranzlmüller, Arndt Bode, Heinz-Gerd Hegering, Henri Casanova and Michael Gerndt, editors, *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC 2009, Garching, Germany, June 11-13, 2009*, pages 49–58. ACM, 2009.

[105] Joshua Hursey, Jeffrey M. Squyres, Timothy Mattox and Andrew Lumsdaine. « The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI ». In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–8. IEEE, 2007.

[106] Parry Husbands, Costin Iancu and Katherine A. Yelick. « A performance analysis of the Berkeley UPC compiler ». In *Proceedings of the 17th International Conference on Supercomputing (ICS'03)*, pages 63–73, 2003.

[107] Eric Roman Jason Duell, Paul Hargrove. « The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart ». Technical Report publication LBNL-54941, Berkeley Lab, 2003.

[108] Morris A. Jette, Andy B. Yoo and Mark Grondona. « SLURM : Simple Linux Utility for Resource Management ». In Dror G. Feitelson, Larry Rudolph and Uwe Schwiegelshohn, editors, *Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'03)*, volume 2862, pages 44–60. Springer-Verlag, 2003.

[109] Laxmikant V. Kalé and Sanjeev Krishnan. « CHARM++ : A Portable Concurrent Object Oriented System Based on C++ ». In A. Paepcke, editor, *Proceedings of The International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'93)*, pages 91–108. ACM Press, September 1993.

[110] Nicholas T. Karonis, Bronis R. de Supinski, Ian Foster, William D. Gropp, Ewing L. Lusk and John Bresnahan. « Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance ». In *14th International Parallel and Distributed Processing Symposium (SPDP'2000)*, pages 377–386, Washington - Brussels - Tokyo, May 2000. IEEE.

[111] Nicholas T. Karonis, Brian R. Toonen and Ian T. Foster. « MPICH-G2 : A Grid-Enabled Implementation of the Message Passing Interface ». *CoRR*, cs.DC/0206040, 2002.

[112] Valentin Kravtsov, David Carmeli, Werner Dubitzky, Krzysztof Kurowski and Assaf Schuster. « Grid-enabling complex system applications with QosCosGrid : An architectural perspective ». In Hamid R. Arabnia, editor, *Proceedings of the 2008 International Conference on Grid Computing & Applications, GCA 2008, Las Vegas, Nevada, USA, July 14-17, 2008*, pages 168–174. CSREA Press, 2008.

[113] Valentin Kravtsov, David Carmeli, Werner Dubitzky, Ariel Orda, Assaf Schuster, Mark Silberstein and Benny Yoshpa. « Quasi-opportunistic Supercomputing in Grid Environments ». In Anu G. Bourgeois and Si-Qing Zheng, editors, *Algorithms and Architectures for Parallel Processing, 8th International Conference, ICA3PP 2008, Cyprus, June 9-11, 2008, Proceedings*, volume 5022 of *Lecture Notes in Computer Science*, pages 233–244. Springer, 2008.

[114] Krzysztof Kurowski, Bogdan Ludwiczak, Jarek Nabrzyski, Ariel Oleksiak and Juliusz Pukacki. « Dynamic grid scheduling with job migration and rescheduling in the GridLab resource management system ». *Scientific Programming*, 12(4) :263–273, 2004.

[115] Krzysztof Kurowski, Mariusz Mamonski, Piotr Grabowski, Yannick Langlois, Guillaume Mecheneau, Thomas Herault, Camille Coti and Mark Ragan. « D1.4 : Second Prototype and Integration of Grid Services Together with QoS-Aware Grid MW Providers ». Technical Report, European Union, QosCosGrid project FP6 IST-2005-033883, October 2008.

[116] Sébastien Lacour, Christian Pérez and Thierry Priol. « A Network Topology Description Model for Grid Application Deployment ». In *GRID*, pages 61–68, 2004.

[117] Ewin Laure and Bob Jones. « Enabling Grids for e-Science : The EGEE Project ». Technical Report EGEE-PUB-2009-001. 1, Sep 2008.

[118] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz and Ben Liblit. « Lessons learned at 208K : towards debugging millions of cores ». In *Proceedings of the International Conference for High Performance Networking Computing, Networking, Storage and Analysis (SC|08)*, page 26, 2008.

[119] Pierre Lemarinier, Aurélien Bouteiller, Thomas Herault, Géraud Krawezik and Franck Cappello. « Improved Message logging versus Improved coordinated checkpointing for fault tolerant MPI ». In *IEEE International Conference on Cluster Computing (Cluster 2004)*. IEEE CS Press, 2004.

[120] Michael Litzkow, Miron Livny and Matthew Mutka. « Condor - A Hunter of Idle Workstations ». In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[121] Michael J. Litzkow, Todd Tannenbaum, Jim Basney and Miron Livny. « Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System ». Technical Report 1346, University of Wisconsin-Madison, 1997.

[122] Pangfeng Liu and Da-Wei Wang. « Reduction Optimization in Heterogeneous Cluster Environments ». In *IPPS : 14th International Parallel Processing Symposium*, pages 477–482. IEEE Computer Society Press, may 2000.

[123] Cyrille Martin. « Déploiement et contrôle d'applications parallèles sur grappes de grandes tailles », December 15 2003.

[124] Moreno Marzolla, Paolo Andreetto, Valerio Venturi, Andrea Ferraro, A. Shiraz Memon, M. Shahbaz Memon, Bastian Tweddell, Morris Riedel, Daniel Mallmann, Achim Streit, Sven van den Berghe, Vivian Li, David F. Snelling, Katerina Stamou, Zeeshan Ali Shah and Fredrik Hedman. « Open Standards-Based Interoperability of Job Submission and Management Interfaces across the Grid Middleware Platforms gLite and UNICORE ». In *eScience*, pages 592–601. IEEE Computer Society, 2007.

[125] Matthew L. Massie, Brent N. Chun and David E. Culler. « The ganglia distributed monitoring system : design, implementation, and experience ». *Parallel Computing*, 30(7) :817–840, July 2004.

[126] Gerald M. Masson. « Binomial Switching Networks for Concentration and Distribution ». COM-25(9), Sept. 1977.

[127] Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Ryousei Takano and Yutaka Ishikawa. « TCP Adaptation for MPI on Long-and-Fat Networks ». In *Proceedings of the 2005 IEEE International Conference on Cluster Computing (CLUSTER'05)*, pages 1–10. IEEE, 2005.

[128] Philip K. McKinley, Yih jia Tsai and David F. Robinson. « Collective Communication in Wormhole-Routed Massively Parallel Computers ». *IEEE Computer*, 28(12) :39–50, 1995.

[129] Hidemoto Nakada, Satoshi Matsuoka, Keith Seymour, Jack Dongarra, Craig Lee and Henri Casanova. « GridRPC : A Remote Procedure Call API for Grid Computing ». In Manish Parashar, editor, *Grid computing — GRID 2002 : third international workshop, Baltimore, MD, USA, November 18, 2002 : proceedings*, volume 2536 of *Lecture Notes in Computer Science*, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 2002. Springer-Verlag. Submitted.

[130] Hidemoto Nakada, Yoshio Tanaka, Satoshi Matsuoka and Satoshi Sekiguchi. « The Design and implementation of a Fault-Tolerant RPC system : Ninf-C ». In *Proceedings of HPC ASIA 2004*, pages pp.9–18, 2004.

[131] Raymond Namyst. « *Contribution á la conception de supports exécutifs multithreads performants* ». Habilitation á diriger des recherches, Université Claude Bernard de Lyon, pour des travaux effectués á l'école normale supérieure de Lyon, DEC 2001.

[132] Jeff Napper and Paolo Bientinesi. « Can Cloud Computing Reach the Top500 ? ». Technical Report AICES-2009-5, Aachen Institute for Computational Engineering Science, RWTH Aachen, February 2009.

[133] Antoine Petitet, Susan Blackford, Jack Dongarra, Brett Ellis, Graham Fagg, Kenneth Roche and Sathish Vadhiyar. « Numerical Libraries And The Grid : The GrADS Experiments With ScaLAPACK ». Technical Report UT-CS-01-460, Innovative Computing laboratory, University of Tennessee, April 2001.

[134] Jelena Pjesivac-Grbovic. « *Automatic and Adaptive Optimizations of MPI Collective Operations* ». Doctorat en sciences, spécialité informatique, The University of Tennessee, Knoxville, December 2007.

[135] Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel and Jack J. Dongarra. « Performance Analysis of MPI Collective Operations ». *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 16 :272a, 2005.

[136] James S. Plank, Micah Beck, Gerry Kingsley and Kai Li. « Libckpt : Transparent Checkpointing under UNIX ». In *USENIX Winter*, pages 213–224, 1995.

[137] James S. Plank, Kai Li and Michael A. Puening. « Diskless checkpointing ». Technical Report CS-97-380, University of Tennessee, December 1997.

[138] James S. Plank, Kai Li and Michael A. Puening. « Diskless Checkpointing ». *IEEE Transactions on Parallel and Distributed Systems*, 9(10) :972–986, October 1998.

[139] Alex Pothen and Padma Raghavan. « Distributed Orthogonal Factorization : Givens and Householder Algorithms ». *SIAM Journal on Scientific and Statistical Computing*, 10 :1113–1134, 1989.

[140] David Powell. « Failure Mode Assumptions and Assumption Coverage ». In *FTCS*, pages 386–395, 1992.

[141] Rolf Rabenseifner. « Automatic MPI Counter Profiling of All Users : First Results on a CRAY T3E 900-512 », 1999.

[142] Rolf Rabenseifner. « Optimization of Collective Reduction Operations ». In Marian Bubak, G. Dick van Albada, Peter M. A. Sloot and Jack Dongarra, editors, *Proceedings of the 4th International Conference on Computational Science (ICCS'04), Part I*, volume 3036 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2004.

[143] Brian Randell. « System structure for software fault tolerance ». In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[144] Ravi Reddy and Alexei Lastovetsky. « HeteroMPI + ScaLAPACK : Towards a ScaLAPACK (Dense Linear Solvers) on Heterogeneous Networks of Computers ». *Proceedings of the 13th IEEE International Conference on High Performance Computing (HiPC 2006)*, 4297 :242–253, 18-21 Dec 2006 2006.

[145] Daniel A. Reed, Charng da Lu and Celso L. Mendes. « Reliability challenges in large systems ». *Future Generation Computer Systems*, 22(3) :293 – 302, 2006.

[146] Ala Rezmerita, Tangui Morlier, Vincent Néri and Franck Cappello. « Private Virtual Cluster : Infrastructure and Protocol for Instant Grids ». In Wolfgang E. Nagel, Wolfgang V. Walter and Wolfgang Lehner, editors, *Proceedings of the 12th European Conference on Parallel and Distributed Computing (EuroPar'06)*, volume 4128 of *Lecture Notes in Computer Science*, pages 393–404. Springer, 2006.

[147] Philip C. Roth, Dorian C. Arnold and Barton P. Miller. « MRNet : A Software-Based Multicast/-Reduction Network for Scalable Tools ». In *Proceedings of the International Conference for High Performance Networking Computing, Networking, Storage and Analysis (SC|03)*, Phoenix, AZ, November 2003. IEEE/ACM SIGARCH. UWisc.

[148] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove and Eric Roman. « The LAM/MPI Checkpoint/Restart Framework : System-Initiated Checkpointing ». In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[149] Richard D. Schlichting and Fred B. Schneider. « Fail Stop Processors : An Approach to Designing Fault-Tolerant Computing Systems ». *ACM Transactions on Computer Systems*, 1 :222–238, 1983.

[150] M Schneider. « Self-stabilization ». *ACM Computing Surveys*, 25(1) :45–67, march 1993.

[151] Steven M. Bellovin. « Defending Against Sequence Number Attacks ». RFC 1948, AT1T Research, May 1996.

[152] Robert E. Strom and Shaula A. Yemini. « Optimistic Recovery in Distributed Systems ». In *Transactions on Computer Systems*, volume 3(3), pages 204–226. ACM, August 1985.

[153] Sun Microsystems, Inc.. « RPC : Remote Procedure Call, Protocol Specification, Version 2 ». RFC 1057, Sun Microsystems, Inc., June 1988.

[154] V. Sunderam. « PVM : A Framework for Parallel Distributed Computing ». *Concurrency : Practice and Experience*, 2(4) :315–339, December 1990.

[155] Ryousei Takano, Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Fumihiro Okazaki and Yutaka Ishikawa. « Effects of packet pacing for MPI programs in a Grid environment ». In *CLUSTER*, pages 382–391. IEEE, 2007.

[156] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura and Satoshi Matsuoka. « Ninf-G : A Reference Implementation of RPC-based Programming Middleware for Grid Computing ». *Journal of Grid Computing*, 1(1) :41–51, 2003.

[157] Yoshio Tanaka, Hiroshi Takemiya, Hidemoto Nakada and Satoshi Sekiguchi. « Design, implementation and performance evaluation of GridRPC programming middleware for a large-scale computational Grid ». In *Proceeding of 5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[158] Todd Tannenbaum, Derek Wright, Karen Miller and Miron Livny. Condor – A Distributed Job Scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[159] Douglas Thain, Todd Tannenbaum and Miron Livny. Condor and the Grid. In Fran Berman, Geoffrey Fox and Tony Hey, editors, *Grid Computing : Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[160] Rajeev Thakur and William D. Gropp. « Improving the Performance of Collective Operations in MPICH ». In Jack Dongarra, Domenico Laforenza and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'03)*, volume 2840 of *Lecture Notes in Computer Science*, pages 257–267, Venice, Italy, October 2003. Springer.

[161] Peter Tröger, Hrabri Rajic, Andreas Haas and Piotr Domagalski. « Standardization of an API for Distributed Resource Management Systems ». In *Proceedings of the 7th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'07)*, pages 619–626. IEEE Computer Society, 2007.

[162] Von Welch, Frank Siebenlist, Ian T. Foster, John Bresnahan, Karl Czajkowski, Jarek Gawor, Carl Kesselman, Sam Meder, Laura Pearlman and Steven Tuecke. « Security for Grid Services ». In *Proceedings of the 12th International Symposium on High-Performance Distributed Computing (HPDC'03)*, pages 48–57, Seattle, Washington, USA, June 2003. IEEE Computer Society.

[163] R. Clint Whaley, Antoine Petitet and Jack J. Dongarra. « Automated Empirical Optimization of Software and the ATLAS Project ». *Parallel Comput.*, 27(1–2) :3–25, 2001.

[164] Asim YarKhan, Jack Dongarra and Keith Seymour. « GridSolve : The Evolution of Network Enabled Solver ». In James C. T. Pool eds. Patrick Gaffney, editor, *Grid-Based Problem Solving Environments : IFIP TC2/WG 2.5 Working Conference on Grid-Based Problem Solving Environments (Prescott, AZ, July 2006)*, pages 215–226. Springer, 2007.

[165] Dantong Yu and Thomas G. Robertazzi. « Divisible Load Scheduling for Grid Computing ». In *in proceedings of the 15th International Conference on Parallel and Distributed Computing and Systems. (PDCS'03)*. Press, 2003.

[166] Tao Zhu, Yongwei Wu and Guangwen Yang. « Scheduling divisible loads in the dynamic heterogeneous grid environment ». In *Proceedings of the 1st International Conference on Scalable Information Systems (Infoscale 2006)", publisher = ACM, editor = Xiaohua Jia, year = 2006, pages = 8, series = ACM International Conference Proceeding Series, volume = 152, isbn = 1-59593-428-6, ee = http ://doi.acm.org/10.1145/1146847.1146855, crossref = DBLP :conf/infoscale/2006, bibsource = DBLP, http ://dblp.uni-trier.de.*

Annexe A

JobProfile de l'application Ray2mesh pour la grille

Listing A.1 – JobProfile d'une application MPI composée de treize processus. Les processus sont répartis dans des groupes eux-mêmes séparés en sous-groupes.

```
<grmsJob appId="example-job">
  <!-- Définition des besoins en ressources de
       calcul et réseaux -->
  <resourceTemplates>
    <networkResourceTemplate templateId="net-high">
      <networkParameter name="bandwidth">
        <min>5120</min>
      </networkParameter>
    </networkResourceTemplate>
    <networkResourceTemplate templateId="net-local">
      <networkParameter name="latency">
        <max>5</max>
      </networkParameter>
    </networkResourceTemplate>

    <computingResourceResourceTemplate templateId="large-memory">
      <hostParameter name="memory">
        <min>4096</min>
      </hostParameter>
    </computingResourceResourceTemplate>
    <computingResourceResourceTemplate templateId="fast">
      <hostParameter name="cpuspeed">
        <min>2.4</min>
      </hostParameter>
    </computingResourceResourceTemplate>
  </resourceTemplates>

 <task taskId="example-task1">
    <resourcesDescription>

      <topology>
      <!-- Définition des groupes de processus -->

        <group groupId="monde">
          <processes>
            <groupIdReference groupId="monde" />
            <!-- Le groupe monde englobe tous les processus -->
            <processesCount>
              <value>$NPROCS</value>
            </processesCount>
          </processes>
          <resourceRequirements>
            <computingResourceResource>
              <templateIdReference templateId="small-memory" />
            </computingResourceResource>
          </resourceRequirements>

          <!-- Définition du maitre -->
          <group groupId="maitre">
            <processes>
```

```
    <groupIdReference groupId="maitre" />
    <processesCount>
      <value>1</value>
    </processesCount>
  </processes>
  <!-- Définition des besoins en ressources de
       calcul du groupe de processus -->
  <resourceRequirements>
    <computingResource>
      <templateIdReference templateId="large-memory" />
    </computingResource>
    <computingResource>
        <templateIdReference templateId="fast" />
    </computingResource>
  </resourceRequirements>
</group> <!-- end maitre -->

<!-- Définition des groupes de premier niveau :
     contremaitre et esclaves, vus directement par
     le maitre -->
<!-- Premier groupe -->
<group groupId="esclaves1">
  <processes>
    <groupIdReference groupId="esclaves1" />
    <processesCount>
      <value>($NPROCS - 1) / 3</value>
    </processesCount>
  </processes>
  <!-- Définition des besoins en ressources
       réseaux à l intérieur du groupe et avec
       les autres groupes de processus -->
  <resourceRequirements>
    <processesConnection>
      <networkResource>
        <templateIdReference templateId="net-local" />
      </networkResource>
    </processesConnection>
  </resourceRequirements>

  <!-- Définition du contremaitre et des esclaves -->
  <group groupId="contremaitre1">
    <processes>
      <groupIdReference groupId="contremaitre1" />
      <processesCount>
        <value>1</value>
      </processesCount>
    </processes>
    <resourceRequirements>
    <!-- Définition des besoins en ressources de
         calcul du groupe de processus -->
      <computingResource>
        <templateIdReference templateId="large-memory" />
      </computingResource>
      <!-- Définition des ressources réseaux pour
           l interconnexion avec le maitre -->
    <groupConnection endpointGroupId="maitre">
      <networkResource>
        <templateIdReference templateId="net-high" />
      </networkResource>
    </groupConnection>
    </resourceRequirements>
  </group> <!-- end contremaitre1 -->

  <group groupId="travailleurs1">
    <processes>
      <groupIdReference groupId="travailleurs1" />
      <processesCount>
        <value>(($NPROCS - 1 ) / 3) - 1</value>
      </processesCount>
    </processes>
    <resourceRequirements>
      <computingResource>
        <templateIdReference templateId="fast" />
      </computingResource>
    </resourceRequirements>
  </group> <!-- end travailleurs1 -->
</group> <!-- end esclaves1 -->


<!-- Deuxième groupe -->
```

```
<group groupId="esclaves2">
  <processes>
    <groupIdReference groupId="esclaves2" />
    <processesCount>
      <value>($NPROCS - 1) / 3</value>
    </processesCount>
  </processes>
  <resourceRequirements>
    <processesConnection>
      <networkResource>
        <templateIdReference templateId="net-local" />
      </networkResource>
  </processesConnection>
  </resourceRequirements>

  <group groupId="contremaitre2">
    <processes>
      <groupIdReference groupId="contremaitre2" />
      <processesCount>
        <value>1</value>
      </processesCount>
    </processes>
    <resourceRequirements>
      <computingResource>
        <templateIdReference templateId="large-memory" />
      </computingResource>
    <groupConnection endpointGroupId="maitre">
      <networkResource>
        <templateIdReference templateId="net-high" />
      </networkResource>
    </groupConnection>
    </resourceRequirements>
  </group> <!-- end contremaitre2 -->

  <group groupId="travailleurs2">
    <processes>
      <groupIdReference groupId="travailleurs2" />
      <processesCount>
        <value>(($NPROCS - 1) / 3) - 1</value>
      </processesCount>
    </processes>
    <resourceRequirements>
      <computingResource>
        <templateIdReference templateId="fast" />
      </computingResource>
    </resourceRequirements>
  </group> <!-- end travailleurs2 -->
</group> <!-- end esclaves2 -->

<!-- Troisième groupe -->
<group groupId="esclaves3">
  <processes>
    <groupIdReference groupId="esclaves3" />
    <processesCount>
      <value>($NPROCS - 1) / 3</value>
    </processesCount>
  </processes>
  <resourceRequirements>
    <processesConnection>
      <networkResource>
        <templateIdReference templateId="net-local" />
      </networkResource>
    </processesConnection>
  </resourceRequirements>

  <group groupId="contremaitre3">
    <processes>
      <groupIdReference groupId="contremaitre3" />
      <processesCount>
        <value>1</value>
      </processesCount>
    </processes>
    <resourceRequirements>
      <computingResource>
        <templateIdReference templateId="large-memory" />
      </computingResource>
    <groupConnection endpointGroupId="maitre">
      <networkResource>
        <templateIdReference templateId="net-high" />
      </networkResource>
```

```
            </groupConnection>
            </resourceRequirements>
          </group> <!-- end contremaitre3 -->

          <group groupId="travailleurs3">
            <processes>
              <groupIdReference groupId="travailleurs3" />
              <processesCount>
                <value>(($NPROCS - 1) / 3) - 1</value>
              </processesCount>
            </processes>
            <resourceRequirements>
              <computingResource>
                <templateIdReference templateId="fast" />
              </computingResource>
            </resourceRequirements>
          </group> <!-- end travailleurs3 -->
        </group> <!-- end esclaves3 -->

      </group> <!-- end monde -->

    </topology>
  </resourcesDescription>

  <execution type="open_mpi">
  <!-- Description de l'application elle-même :
       arguments, fichiers de données à fournir en entrée, redirection des
       entrées/sorties...-->
    <executable>
      <execFile>
        <file>
          <url>gsiftp://myhost.domain.com/~/maitre-esclave</url>
        </file>
      </execFile>
    </executable>
    <arguments>
      <value>dataFilename</value>
    </arguments>
    <inputFile>
      <url>gsiftp://myhost.domain.com/~/input.dat</url>
    </inputFile>
    <processesCount>
      <value>5</value>
    </processesCount>
    <stdout>
      <url>gsiftp://myhost.domain.com/~/example.out</url>
    </stdout>
    <stderr>
      <url>gsiftp://myhost.domain.com/~/~example.err</url>
    </stderr>
  </execution>

 </task>
</grmsJob>
```

Annexe B

Algorithme de sélection des modules dans OpenMPI

Listing B.1 – Algorithme de chargement des modules uniques dans MCA

```
void* selection_composant( liste* liste_des_composants ) {
  module_t *module_courant, *module_choisi;
  int priorité, priorité_max;
  liste* liste_copie;
  liste_copie = copie( liste_des_composants );
  /* charger et initialiser les modules, choisir celui de plus haute priorité */
  tant que ( NULL != liste_des_composants ) {
    module_courant = charger( liste_des_composants -> courant );
    enlister( liste_des_composants -> courant -> modules, module_courant );
    si( NULL != module_courant ) {
      priorité = module_courant -> initialiser();
      si( priorité > priorité_max ){
        priorité_max = priorité;
        module_choisi = module_courant;
      }
    }
    liste_des_composants = liste_des_composants -> suivant;
  }
  si ( NULL == module_choisi ) {
    retourner ERREUR;
  } sinon {
    retourner module_choisi;
  }
}
```

Listing B.2 – Algorithme amélioré de chargement des modules uniques dans MCA

```
void* selection_composant( liste* liste_des_composants ) {
  module_t *module_courant, *module_choisi;
  int priorité, priorité_max;
  liste* liste_copie;
  liste_copie = copie( liste_des_composants );
  /* charger et initialiser les modules, choisir celui de plus haute priorité */
  tant que ( NULL != liste_des_composants ) {
    module_courant = charger( liste_des_composants -> courant );
    enlister( liste_des_composants -> courant -> modules, module_courant );
    si( NULL != module_courant ) {
      priorité = module_courant -> initialiser();
      si( priorité > priorité_max ){
        priorité_max = priorité;
        module_choisi = module_courant;
      }
    }
    liste_des_composants = liste_des_composants -> suivant;
  }
  /* décharger les modules non utilisés */
  tant que ( NULL != liste_copie ) {
    liste* modules = liste_copie -> courant -> modules;
    tant que( NULL != modules -> courant ) {
      si( modules -> courant != module_choisi ) {
        décharger( modules -> courant );
      }
      modules -> courant = modules -> suivant;
    }
  }
  si ( NULL == module_choisi ) {
    retourner ERREUR;
  } sinon {
    retourner module_choisi;
  }
}
```

Résumé

Un environnement de programmation MPI est constitué de trois composants principaux :
- un système de lancement, qui déploie l'application sur les ressources de calcul distantes disponibles ;
- un environnement d'exécution, qui démarre et finalise l'application, contrôle son exécution et son état, et met en relation les processus de l'application ;
- une bibliothèque de communications, qui permet aux processus d'échanger des données au cours de l'exécution.

Le système de lancement peut être un composant externe à l'environnement de programmation et être seulement interfacé avec l'environnement d'exécution, ou faire partie de l'environnement d'exécution.

Le rôle de l'environnement d'exécution est de rendre des services à l'application durant son exécution. Le premier de ces services concerne le cycle de vie de l'application, par son déploiement, son lancement et sa terminaison, et, durant l'exécution, la surveillance de son état et le comportement à suivre en cas de défaillance. L'autre service rendu par l'environnement d'exécution consiste à mettre en relation les processus de l'application pour leur permettre de communiquer en utilisant la bibliothèque de communications.

On peut alors décomposer ses fonctionnalités en trois catégories : le déploiement et le lancement de l'application, les communications internes à l'environnement d'exécution (collectives et point-à-point), et la détection de défaillances.

Au cours de cette thèse, j'ai étudié des problématiques posées par la grande échelle aux environnements d'exécution pour applications parallèles communiquant par passage de messages suivant le paradigme MPI.

Dans un premier temps, j'ai étudié le passage à l'échelle de l'environnement d'exécution lui-même, à travers les performances de ses fonctionnalités principales : le lancement d'applications, et les communications internes.

Les défaillances étant inévitables dans un système à grande échelle, j'ai ensuite étudié des mécanismes de tolérance aux pannes, nécessitant notamment une participation active de l'environnement d'exécution.

Enfin, j'ai étudié un type particulier de systèmes à grande échelle avec les grilles de calcul formées par agrégation de grappes, en proposant un environnement de communications MPI adapté aux communications sur grilles en termes d'impératifs de sécurité et reposant sur un environnement d'exécution permettant d'utiliser des techniques d'interconnexion avancées de passage de pare-feux.

Cet environnement d'exécution et de communications s'intègre dans une pile logicielle permettant la programmation d'applications sur grilles utilisant une nouvelle méthode d'adaptation à la topologie et mettant à contribution le meta-ordonnanceur de grilles. Dans la dernière partie de cette thèse, j'ai étudié comment des applications pouvaient adapter leurs schémas de communications et de calculs à la topologie de la grille en utilisant les fonctionnalités de cet environnement d'exécution.

Mots clés : environnement d'exécution distribué, applications parallèles, grilles de calcul, grande échelle

Abstract

An MPI programming environment is made of three main components :
- a launching system which is in charge of deploying the application on the remote available computing resources ;
- a run-time environment which controls the start-up and finalization fo the application, its execution and its state, and allows the application's processes to communicate with one another ;
- a communication library that allows the processes to exchange data during the execution.

The launching system can be a third-party piece of software, external to the run-time environment and only interfaced with it, or it can be part of it.

The run-time environment is meant to provide services to the application during its execution (at run-time). The first one is about controling the life cycle of the application, namely, deploying, starting-up and finalizing it qnd during the execution monitoring its state of health and following a given behavior upon failures. The other service provided by the run-time environment consists of enabling communications between processes through the communication library.

Run-time environments' functionalities can be decomposed into three categories : deployment and application start-up, internal communications (collective and point-to-point communications) and failure detection.

Througout this thesis, I studied problematics raised by large scale for run-time environments for parallel, message passing applications following the MPI communication paradigm.

I first studied the scalability of the run-time environment itsalf through the performance of its main functionalities : application start-up and internal communications.

Since failures cannot be dodged in large-scale systems, I then studied tault tolerance mechanisms and more specifically, what support must be provided by the run-time environment to support actively those mechanisms.

Last, I studied a special kind of large-scale systems with computational grids made out of an aggregation of clusters. I proposed an adapted run-time environment for MPI communication throughout grids taking into acount security requirements such as firewalls and network address translation and using advanced connectivity techniques to bypass these security devices.

The run-time and communication environment is integrated in a software stack that provides a possibility to program applications for grids using a novel method for adapting the application to a given topology and asking the metascheduler for its contribution to making the physical topology match the application's communication patterns. In the last part of this thesis I studied how applications can use adapted communication and computation patterns for the grid using the features provided by this run-time environment.

Keywords : distributed run-time environment, parallel applications, computational grids, large scale