Ordonnancement des sauvegardes/reprises d'applications de calcul haute performance dans les environnements dynamiques

Blaise Omer Yenke

UNIVERSITÉ DE GRENOBLE
UNIVERSITÉ DE YAOUNDE I

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique : Systèmes et Logiciels**

Arrêté ministérial : 6 janvier 2005

Présentée par

YENKE BLAISE OMER

Thèse dirigée par **Jean-François MEHAUT**
et codirigée par **Maurice TCHUENTE**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **Mathématiques, Sciences et Technologies de l'Information, Informatique**

Ordonnancement des sauvegardes/reprises d'applications de calcul haute performance dans les environnements dynamiques

Thèse soutenue publiquement le **10 Janvier 2011**,
devant le jury composé de :

Professeur, Gabriel NGUETSENG
Univ. Yaounde I, Président
Professeur, Christophe CERIN
Univ. Paris XIII, Rapporteur
Professeur, Laurent PHILIPPE
Univ. Franche-Comté, Rapporteur
Professeur, Claude TANGHA
Univ. Yaounde I, Examinateur
Professeur, Jean-François MEHAUT
Univ. Grenoble, Directeur de thèse
Professeur, Maurice TCHUENTE
Univ. Yaounde I, Co-Directeur de thèse

Résumé. Ordonnancement des sauvegardes/reprises d'applications de calcul haute performance dans les environnements dynamiques

Les avancées technologiques ont conduit les grandes organisations telles que les entreprises, les universités et les instituts de recherche à se doter d'intranets constitués de plusieurs serveurs et d'un grand nombre de postes de travail. Cependant dans certaines de ces organisations, les postes de travail sont très peu utilisés pendant la nuit, les week-ends et les périodes de congés, libérant ainsi une grande puissance de calcul disponible et inutilisée.

Dans cette thèse, nous étudions l'exploitation de ces temps de jachère afin d'exécuter des applications de calcul haute performance. A cet effet, nous supposons que les postes acquis sont rebootés et intégrés à des grappes virtuelles constituées dynamiquement. Toutefois, ces temps de jachère ne permettent pas toujours d'exécuter les applications jusqu'à leur terme. Les mécanismes de sauvegarde/reprise (checkpointing) sont alors utilisés pour sauvegarder, dans un certain délai, le contexte d'exécution des applications en vue d'une éventuelle reprise. Il convient de noter que la sauvegarde de tous les processus dans les délais impartis n'est pas toujours possible. Nous proposons un modèle d'ordonnancement des sauvegardes en parallèle, qui tient compte des contraintes temporelles imposées et des contraintes liées aux bandes passantes (réseau et disque), pour maximiser les temps de calcul déjà effectués pour les applications candidates à la sauvegarde.

Mots clés : Ordonnancement, sauvegarde/reprise d'applications, grappes virtuelles, gestionnaire de ressources

Abstract. Scheduling checkpoint/restart of high performance computing on dynamic environments

The technological advances has led major organizations such as enterprises, universities and research institutes to acquire intranets consisting of several servers and many workstations. However, in some of these organizations, the resources are rarely used at nights, weekends and on holidays, thus releasing a large computing power available and unused.

This thesis discusses the exploitation of the idle period of workstaions in order to run HPC applications. The workstations retained are restarted and integrated in dynamically formed clusters. However, the idle periods do not always permit the complete carrying out of the computations allocated to them. The checkpointing mechanisms are then used to save in a certain period, the execution context of applications for a possible restart. It is worth nothing that checkpointing all the processes in the required period is not always possible. We propose a scheduling model of checkpointing in parallel, which takes into account the time constraints imposed and the bandwidth constraints (network and disk) to maximize the computation time already taken for the applications which are to be checkpointed.

Keywords : Scheduling, Checkpointing of applications, virtual clusters, batch schedulers

Je rends infiniment grâce à Dieu, le Seigneur Tout Puissant, pour la santé , le courage et la persévérance qu'il m'a accordés pour mener à terme cette thèse.

J'adresse mes remerciements en tout premier lieu à mes deux directeurs de thèse, Maurice Tchuenté qui a initié ce co-encadrement et Jean-François Méhaut qui a bien voulu m'accueillir dans le laboratoire LIG de Grenoble. Un des atouts d'une thèse en co-tutelle est de pouvoir bénéficier de l'expérience de chacun des directeurs de thèse. J'ai ainsi été enrichi sur le plan scientifique des compétences venant de deux fortes personnalités, toutes deux évoluant dans deux domaines privilégiés, ayant chacune sa perception des choses, mais étant toutes deux animées d'une forte passion pour la recherche. Mes deux directeurs ont su orienter mes recherches et me soutenir en temps opportun. Pour tout cela, je leur suis infiniment reconnaissant.

Je voudrais associer à cette reconnaissance le Professeur Brigitte Plateau, Directeur du Laboratoire d'Informatique de Grenoble, pour l'intérêt qu'elle a toujours porté à mes travaux.

Je tiens également à adresser un grand merci à mon jury et notamment : Gabriel Nguetseng pour m'avoir fait l'honneur de présider le jury de ma soutenance ; Christophe Cérin, Franck Capello et Claude Tangha, pour avoir accepté de consacrer une partie de leur temps à rapporter sur ces travaux.

Une thèse est l'occasion de multiples collaborations, à la fois scientifiques et humaines. À défaut de pouvoir dresser une liste exhaustive, je voudrais remercier :

- Elvis Houpa pour son expertise mathématique et ses suggestions tant sur le fond que sur la forme de ce travail ;
- Serge Moto pour sa disponibilité et sa contribution à la relecture de ce document ;
- Jean Michel Nlong II, pour ses remarques pertinentes qui ont contribué a améliorer la qualité de ce document ;
- Martin Mbeuyo, mon beau-frère, pour les enrichissants échanges que nous avons partagés ;
- les membres du laboratoire LIG que j'ai eu le plaisir de côtoyer et qui n'ont pas hésité à répondre à mes sollicitations. Un clin d'oeil particulier à Christian Seguy pour sa grande disponibilité ;
- les thésards du Brésil et de la Tunisie, qui m'ont permis de découvrir d'autres horizons culturels.

Un grand merci à ma famille – ma mère Yenké Nguengtat Pauline, mon oncle Keutchankeu François qui est un second père pour moi, mes frères et sœurs Keutchaya Charly, Tchana Yenké Sylvain, Djakou Sidonie, Djeubia Annette et Yonkeu Irène pour leur soutien, leur aide et leur gentillesse avant, pendant et après cette thèse.

Je termine par celle qui illumine ma vie depuis maintenant plus de 10 ans, ma tendre épouse Hermine. Merci pour ton soutien, ton courage, ta bravoure, ton immense gentillesse qui me donnent toute la motivation d'aller de l'avant et de faire toujours mieux. Toi et nos quatre trésors Ahianor, Manuela, Annette et Blaise Pascal êtes ce que j'ai de plus précieux.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Durant les deux dernières décennies, les systèmes informatiques ont connu un progrès remarquable. Cette évolution a été marquée par deux avancées technologiques importantes : le développement des microprocesseurs de plus en plus puissants qui ont conduit à la construction des ordinateurs performants et à coûts réduits, et le développement des réseaux informatiques très rapides. En conséquence, les grandes organisations (les entreprises, les instituts de recherche, les universités, etc.) disposent aujourd'hui d'intranets dotés de serveurs et d'un grand nombre de postes de travail.

Dans certaines de ces organisations, les postes de travail sont très peu utilisés la nuit, les week-ends et pendant les périodes de congés. Si l'on considère par exemple que le personnel de ces organisations est présent la journée de 8h00 à 12h00 et de 14h00 à 18h00, alors sur une semaine normale, un poste de travail est inutilisé pendant près de 128 heures, soit l'équivalent de plus de 5 jours, ceci libérant une grande puissance de calcul. Il est donc judicieux d'exploiter les longues périodes d'inactivité des postes de travail pour exécuter des applications nécessitant la ressource de calcul.

Dans cette thèse, nous adoptons de constituer des grappes virtuelles à partir des postes non utilisés pour exploiter à bon escient les temps de jachère, comme l'avait envisagé le projet IGGI [1]. Une grappe virtuelle est une infrastructure de calcul constituée de ressources réparties sur un intranet de façon dynamique. Sur des architectures de type grappe virtuelle, la volatilité des ressources est une des propriétés à prendre en compte. Les postes de travail doivent par exemple être restitués le matin au retour des employés. Il y a donc nécessité d'assurer la sauvegarde des calculs inachevés, en vue d'une éventuelle reprise. Compte tenu de la volatilité des ressources, la sauvegarde en local d'un calcul inachevé n'est pas indiquée, car il n'y a aucune garantie que la ressource de calcul soit disponible le lendemain. Par conséquent, pour ces architectures de grappes virtuelles, il est adéquat d'effectuer les sauvegardes sur le disque du serveur de l'intranet.

Les mécanismes de sauvagarde/reprise de contexte des processus sont un sujet important pour la reprise des applications. En effet, supposons qu'un utilisateur exécute une application ayant un long temps de calcul (disons 3 jours) dans une grappe virtuelle constituée le week-end (disponibilité : 2 jours). En l'absence des mécanismes de sauvegarde/reprise (checkpoint/restart

ou tout simplement checkpointing), l'utilisateur perdra tout le volume de calcul déjà effectué (soit près de 2 jours de calcul) au moment de la libération des ressources utilisées pour constituer la grappe virtuelle. Par contre si l'utilisateur avait la possibilité de sauvegarder le contexte d'exécution de son processus, lorsque la grappe redeviendrait disponible, il redémarrerait son programme à partir de la dernière sauvegarde effectuée. Les mécanismes de sauvagarde/reprise sont donc d'une utilité certaine. En revanche ils ont un coût en temps et en espace mémoire.

Dans une infrastructure de grappe virtuelle où plusieurs applications s'exécutant simultanément ont à se retirer pour libérer les ressources, sachant que le réseau et le disque sont partagés entres les différentes applications, on peut faire face à des contraintes liées aux bandes passantes. De plus si l'on considère que les applications doivent être sauvegardées pendant un certain délai avant la libération des ressources, on n'est pas sûr de pouvoir les sauvegarder toutes. Il se pose alors la question de savoir s'il ne serait pas opportun de sauvegarder certaines applications et pas d'autres. Ces observations conduisent ainsi à un problème d'évaluation de performance du dispositif de sauvegarde et à un problème d'ordonnancement et d'optimisation.

1.1 Objectif

L'objectif de ce travail est de proposer un mécanisme de sauvegarde d'applications, afin de maximiser les temps de calcul effectués avant une libération imposée de ces ressources qui ont été acquises dans un intranet pour constituer une grappe virtuelle. Il s'agit d'une solution à un problème d'ordonnancement avec une contrainte de temps et de bande passante.

Ce travail vise donc à doter les environnements de calcul haute performance d'un outil pour accroître la qualité des services offerts aux utilisateurs finaux.

1.2 Contributions

L'étude des techniques de sauvegarde/reprise des applications dans les environnement parallèles et répartis n'est pas un problème nouveau [2, 3, 4, 5, 6, 7]. L'originalité de notre travail repose d'une part sur la proposition d'une estimation de la bande passant nécessaire pour la sauvegarde en parallèle d'un certain nombre d'applications sans perte de performance du dispositif de sauvegarde, et d'autre part sur la proposition d'une formulation toute nouvelle du problème du choix des applications à sauvegarder dans un certain délai. Ceci a conduit à la proposition d'un algorithme d'ordonnancement des sauvegardes et au développement d'un prototype intégré à un gestionnaire de ressources.

De manière plus détaillée, les contributions apportées par ce travail de recherche sont les suivantes :

– Nous proposons une fonction bw qui donne la bande passante $bw(m, V)$ nécessaire pour la sauvegarde en parallèle de m applications de volume mémoire agrégé V. Cette fonction est utilisée pour estimer le temps de sauvegarde de chacune des m applications, dans un contexte où la bande passante $bw(m, V)$ est équitablement partagée entre les m tâches.

bw est une fonction polynômiale de degré deux en m et V, ce qui lui confère de bonnes propriétés. Cette fonction est utilisée pour calibrer le nombre d'applications que l'on peut sauvegarder sans perte de performance du dispositif de sauvegarde.
- Pour le problème du choix des applications à sauvegarder dans un certain délai, nous proposons une formulation basée sur une nouvelle variante du problème du sac-à-dos, et nous démontrons que ce nouveau problème est NP-complet.
- Nous proposons ensuite comme solution approchée, un algorithme d'ordonnancement où la sélection des candidats se fait en deux phases. Dans la première phase l'algorithme utilise la fonction *bw* pour déterminer de façon gloutonne les candidats qui peuvent être sauvegardés en parallèle sans perte de performance du système de sauvegarde. Dans la seconde phase, l'algorithme utilise l'approche semi-énumérative présentée dans [8] pour le problème de sac à dos de petite taille pour déterminer les tâches à sauvegarder en tenant compte de la contrainte de délai.
- Nous proposons enfin l'intégration de notre algorithme d'ordonnancement des sauvegardes dans un gestionnaire de ressources. Plus précisément, nous présentons l'intégration au gestionnaire de ressources OAR [9]. Le prototype implémenté est une application client/-serveur (basée sur le modèle *un thread par client accepté* [10]) qui exploite les données provenant de la base de données du gestionnaire de ressources de la grappe pour faire l'ordonnancement des sauvegardes. Par exemple, pour le cas de l'intégration à OAR, nous présentons l'interaction entre le prototype implémenté de l'ordonnanceur des sauvegardes avec la base de données de OAR. Ce prototype est portable et peut être aisément modifié pour interagir avec tout autre gestionnaire de ressources, du moment que la structure de la base de données est connue.

Cette thèse a conduit la publication de :
- 1 article dans une conférence francophone avec proceedings et reviews [11]
- 1 article dans une conférence internationale avec proceedings et reviews [12], article qui a obtenu le **best paper award**
- 1 article à paraître dans le journal international IEEE Transaction on Service Computing [13]
- 1 article dans une conférence internationale avec proceedings et reviews [14]

1.3 Cadre de travail

Cette thèse est le fruit d'une co-tutelle entre l'Université de Yaoundé 1 et l'Université Joseph Fourier de Grenoble. Les travaux sur grilles et grappes ont été effectués pour l'essentiel au laboratoire LIG (Laboratoire d'Informatique de Grenoble), au sein du projet MESCAL (Middleware Efficiently SCALable) [15], un projet commun CNRS, INPG, INRIA et UJF. Le but de MESCAL est de concevoir et de valider les intergiciels et les services qui permettent d'exploiter efficacement les grandes infrastructures de calcul. Les applications visées concernent essentiellement le calcul scientifique haute performance. La conception de systèmes qui passent à l'échelle, par modélisation et évaluation des performances des architectures, des couches logicielles, ainsi que des applications, est l'un des axes de recherche de MESCAL.

1.4 Organisation du manuscrit

La suite de ce document est organisée en trois parties.
La **première partie**, incluant les chapitres 2 et 3, présente le problème de la sauvegarde d'applications de calcul haute performance dans les environnements dynamiques.

- Le chapitre 2 présente les environnements de calcul de type grappe ainsi que les différentes approches de leur mise en place. Il introduit également les principales approches d'exploitation des ressources libres des intranets pour en faire des grappes de calcul. Le choix d'une approche adaptée à notre contexte termine ce chapitre.
- Les mécanismes de sauvegarde/reprise d'applications sont abordés au chapitre 3 où les différentes techniques utilisées et quelques systèmes implémentant ces techniques sont présentés. Ce chapitre présente certains paramètres importants à prendre en compte pour l'étude des performances des mécanismes de sauvegarde/reprise. Différentes stratégies de sauvegarde sont ensuite abordées. Ce chapitre s'achève sur l'adoption de la sauvegarde en parallèle qui cadre avec le souci d'utiliser à bon escient les ressources libres d'un intranet, tout en mettant en évidence le problème des goulots d'étranglement qui en découlent et dont il faut tenir compte.

La **deuxième partie** (chapitres 4 et 5) se concentre sur la présentation du modèle d'ordonnancement des sauvegardes d'applications que nous proposons.

- Cette partie commence au chapitre 4 par l'analyse des différents paramètres qui interviennent dans la sauvegarde en parallèle d'applications qui partagent le réseau et le disque du serveur. Cette analyse montre la pertinence de ces paramètres qui rend complexe le problème de la sauvegarde en parallèle d'applications. Sur la base d'un grand nombre d'expérimentations menées, une proposition est faite pour l'estimation de la bande passante du système nécessaire pour la sauvegarde en parallèle d'un certain nombre d'applications séquentielles sans que les performances du réseau et du disque ne se dégradent. Le chapitre s'achève sur l'analyse de la sauvegarde en parallèle d'applications parallèles et relève la complexité d'établir un modèle de prédiction dans ce cas.
- Au chapitre 5, une proposition d'une approche pour l'ordonnancement avec contraintes des sauvegardes d'applications dans les environnements dynamiques comme les grappes virtuelles est présentée. Une formulation du problème d'ordonnancement est présentée et on démontre que c'est un problème NP-complet. Le chapitre s'achève sur la proposition d'un algorithme d'ordonnancement qui produit des solutions approchées.

La **troisième partie**, (chapitres 6 et 7) présente une implémentation de notre algorithme d'ordonnancement et son intégration dans un gestionnaire de ressources.

- Le chapitre 6 présente les différents éléments d'implémentation de l'ordonnanceur et son intégration dans un gestionnaire de ressources.
- Le chapitre 7 présente l'évaluation de l'intégration de l'ordonnanceur des sauvegardes dans un gestionnaire de ressources. Cette évaluation est faite avec quelques exemples menées sur GRID5000.

Le manuscrit s'achève par le chapitre 8 qui rappelle le problème posé, les solutions que nous proposons, résume les apports essentiels et présente des perspectives ouvertes par ce travail de recherche.

Première partie

Sauvegarde/Reprise d'applications dans les environnements dynamiques : État de l'art

Chapitre 2

Environnements dédiés et dynamiques : grappes, grappes virtuelles

Sommaire

L'objectif de ce chapitre est de présenter les environnements de calcul de type grappe ainsi que les différentes approches de leur mise en place. Nous y présentons les principales approches d'exploitation des ressources libres des intranets pour en faire des grappes de calcul. Enfin, dans la dernière section, nous précisons l'approche choisie dans le cadre de notre travail ainsi que les contraintes et hypothèses sous-jacentes.

2.1 Architectures de machines

Les besoins en puissance de calcul informatique dans divers domaines (calcul scientifique, prévision météorologique et climatique, fouille de données, simulation moléculaire) sont sans cesse croissants. Deux types d'architectures de machines tentent d'apporter une solution à ces sollicitations : les machines parallèles et les environnements distribués. Afin de mieux cerner ces notions, nous présentons tout d'abord dans la section qui suit, la classification introduite par Flynn pour les architectures de machines.

2.1.1 Classification de Flynn

La classification des architectures de machines en quatre catégories proposée par Flynn [16], est basée sur les notions de flot de données et de flot d'instructions. Suivant cette taxinomie, les quatre principaux types de machines sont les suivants :

- les machines SISD (*Single Instruction Single Data*) : dans ces machines, une seule instruction est exécutée et une seule donnée est traitée à tout instant. Dans ce type de machine, les traitements sont séquentiels. Ce modèle correspond à une machine monoprocesseur monocœur, typiquement la machine de Von Neuman. Il est important de noter que ce type de machine n'est plus produit depuis deux décennies.
- les machines SIMD (*Single Instruction Multiple Data*) : ces machines contiennent n processeurs tous identiques et sont contrôlées par une seule unité de contrôle centralisée. A chaque étape, tous les processeurs exécutent la même instruction de façon synchrone mais sur des jeux de données différents. Ces machines, aussi appelées machines vectorielles, sont utilisées pour les calculs spécialisés. Les processeurs de telles machines correspondent par exemple aux unités de calcul en virgule flottante (*Floating Point Unit*, FPU [1]). Un exemple de machine actuelle de type SIMD est le supercalculateur vectoriel NEC SX-9 [2] constitué de noeuds SMP (Symetric MultiProcessors). Dans cette machine, chaque noeud est constitué de 16 CPUs et délivre une puissance de 1.6 Teraflops.
- les machines MISD (*Multiple Instructions Single Data*) : ces machines peuvent exécuter plusieurs instructions sur la même donnée de façon synchrone. Ce modèle englobe certaines architectures de type pipeline et les architectures tolérant les pannes par réplication de calcul.
- les machines MIMD (*Multiple Instructions Multiple Data*) : dans ce modèle, chaque processeur est autonome, dispose de sa propre unité de contrôle et exécute son propre flot

1. pagesperso-orange.fr/aedvlsi/PDF/AED_fpu.pdf
2. http ://www.necam.com/SX/

d'instructions sur son propre flot de données. Des informations plus détaillée sont fournies sur ces types de machines avec la classification de A. Tanenbaum [17] qui, par ce groupe, distingue 4 sous-classes en se basant sur les interconnexions réseaux (Figure 2.1).

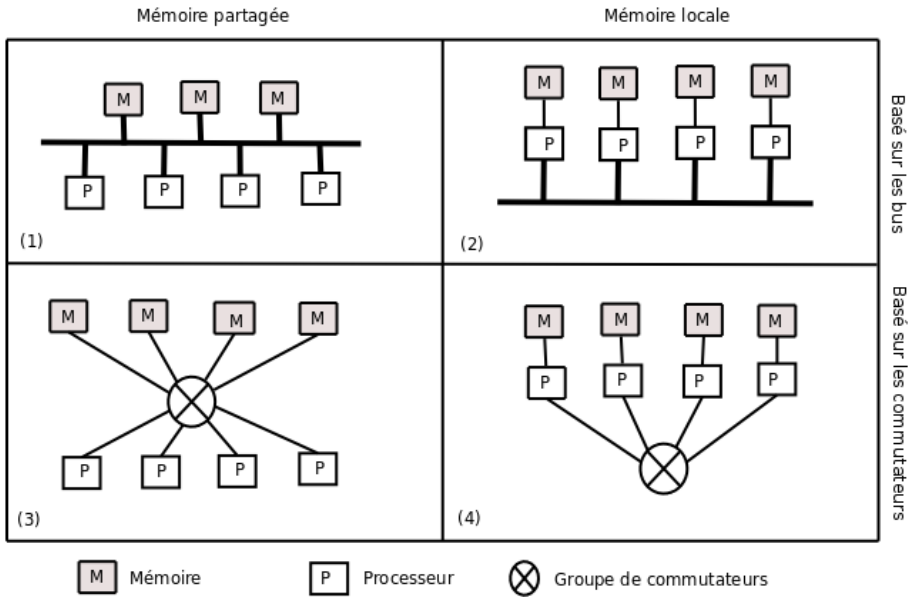

FIGURE 2.1 – Classification de Tanenbaum des architectures de machines de type MIMD

La classification de Tannenbaum met en exergue les deux architectures des machines les plus utilisées aujourd'hui pour résoudre les problèmes nécessitant de grandes ressources de calcul : les architectures multiprocesseurs à mémoire partagée et les architectures multiprocesseurs à mémoire distribuée. Les performances de ces architectures de machines sont sans cesse croissantes avec l'avènement des processeurs multicœurs.

2.1.2 Processeurs multicœurs

Les besoins de plus en plus élevés en puissance de calcul avaient amené les constructeurs à augmenter la puissance des processeurs monocœurs en y élevant la fréquence d'horloge. De ce fait, les processeurs devenaient de plus en plus puissants. Cependant, cette augmentation de fréquence au sein des processeurs nécessitait l'accroissement de la puissance électrique consommée, ce qui générait une énergie thermique qu'il fallait dissiper. Malgré cette augmentation de fréquence, les processeurs monocœurs était donc limités dans la puissance qu'ils pouvaient offrir. Ceci a amené les constructeurs à se lancer dans la fragmentation des processeurs, ce qui a abouti à la création des processeurs multicœurs.

Un processeur multicœurs est composé d'au moins deux unités de calcul (cœurs) gravées au sein d'une même puce. Dans la plupart des cas, les cœurs des processeurs sont homogènes et fonctionnent à la même fréquence que ces processeurs. Cette reproduction du parallélisme au sein d'un unique processeur permet aux processeurs multicœurs d'avoir une puissance supérieure

à celle des processeurs monocœurs, avec une fréquence d'horloge moins élevée réduisant de ce fait la consommation d'énergie et la chaleur dégagée.

Parmi les différents facteurs qui entraînent les changements importants qui se produisent actuellement dans la conception des microprocesseurs et des systèmes haute performance, les deux qui suivent sont particulièrement remarquables : 1) le nombre de transistors sur une puce continue à doubler tous les 18 mois environ, mais la vitesse d'horloge des processeurs ne va pas continuer à augmenter ; 2) le nombre de broches et la bande passante sur les processeurs sont entrain d'atteindre leurs limites [18]. De ce fait le nombre de cœurs par processeur atteindra très rapidement un seuil limite. Ainsi, pour tirer parti des avantages offerts par l'architecture de processeurs multicœurs dans des environnements de calcul haute performance, il est important de concevoir de nouvelles architectures de machines multiprocesseurs et il va falloir développer de nouveaux modèles de programmation.

Les processeurs multicœurs offrent plus de parallélisme avec encore plus de puissance. Par exemple si un processeur monocœur cadencé à 1,5 GHz et ayant une puissance de 20GFlops est remplacé par un processeur bi-cœur cadencé à 1,5 GHz et dont la puissance de chaque cœur vaut 20 GFlops alors la puissance total du processeur bi-cœur vaudra 40 GFlops. Cependant, le défi est de pouvoir développer des applications qui puissent être efficacement opérationnelles dans ces architectures.

2.1.3 Architectures multiprocesseurs à mémoire partagée

Les architectures multiprocesseurs à mémoire partagée sont connues sous le nom de machines parallèles. Une machine parallèle (communément appelée *supercalculateur*) est une plateforme homogène, dont toutes les composantes présentent la même architecture, à l'intérieur d'un seul domaine d'administration. Les types récents de machines parallèles, caractérisées principalement par différents modèles d'interconnexion entre les processeurs et la mémoire, sont : les architectures de type SMP qui permettent à un nombre relativement restreint de processeurs (inférieur à 32) de partager une mémoire commune et l'architecture CC-NUMA qui est une extension de l'architecture SMP. Elle permet à un nombre plus élevé de processeurs de communiquer par mémoire partagée.

- **SMPs** (*Symetric MultiProcessors*) ou **cc-UMAs** (*Cache Coherence Uniform Memory Access*). Cette architecture est composée de plusieurs processeurs identiques qui accèdent à une mémoire partagée (Figure 2.1 (1)). Le temps d'accès mémoire est identique entre processeurs. Cependant, l'accès à la mémoire constitue un goulot d'étranglement sur ce type d'architecture dès que le nombre de processeurs devient important.
- **CC-NUMAs** (*Cache Coherence Non-Uniform Memory Access*). Sur cette architecture (Figure 2.1 (3)), chaque processeur ou groupe de processeurs dispose de sa propre mémoire vive à laquelle il peut accéder rapidement. Les processeurs communiquent entre eux par un système de communication hiérarchique qui les relie. Contrairement aux SMPs, le temps d'accès mémoire n'est pas uniforme. L'accès à la mémoire d'un autre processeur est moins rapide qu'un accès local.

De nouvelles machines parallèles de plus en plus puissantes sont proposées sur le marché par les constructeurs. Ces machines sont constituées de centaines (voire milliers) de processeurs multicœurs.

L'évolution des technologies des architectures est suivie par le TOP500 [3] qui effectue un classement biannuel des 500 machines les plus puissantes de par le monde. La puissance de calcul est évaluée à partir du benchmark d'algèbre linéaire LINPACK. Le leader de ce classement en date du 11 novembre 2009 est la plate-forme *Jaguar* [4] avec une puissance de calcul évaluée à 1,75 petaflops. Cette super-machine CRAY XT5 détrône ainsi le supercalculateur *Roadrunner* d'IBM qui était le premier à passer le pic du petaflops avec une puissance de 1,04 petaflops en juin 2008.

Le côté onéreux des supercalculateurs restreint cependant leurs usages à quelques laboratoires appartenant à de très grosses entreprises qui ont la possibilité de les acquérir.

2.1.4 Architectures multiprocesseurs à mémoire distribuée

La montée en puissance des simples PCs peu chers et des réseaux d'interconnexion haute performance a permis de concevoir une autre approche qui consiste à connecter des machines à bas prix que l'on trouve sur le commerce par des réseaux classiques. Ces nouvelles architectures portent le nom de systèmes distribués (ou répartis). Un système distribué est un ensemble d'ordinateurs indépendants qui, grâce à leur interconnexion et au gestionnaire de ressources qui les administre, apparaît à un utilisateur comme un système unique et cohérent. Dans cette classe de machines (Figure 2.1 (4)), on retrouve les architectures multiprocesseurs à mémoire distribuée à moyenne échelle telles que les intranets et les grappes, les architectures multiprocesseurs à mémoire distribuée à grande échelle telles que Internet et les grilles.

Intranets

Un intranet est une interconnexion d'ordinateurs standards qui offre un ensemble de services internes à un réseau local. L'architecture d'un intranet est basée sur le modèle client-serveur, où les clients et les serveurs communiquent à travers les protocoles TCP/IP et HTTP. Les intranets d'organisations sont généralement connectés au réseau mondial Internet via des passerelles et des pare-feux pour des raisons de sécurité.

Les activités menées au sein d'un intranet d'une organisation sont généralement la mise à disposition d'informations sur l'organisation auprès des usagers, la messagerie et les échanges de données entre usagers, la recherche et la consultation de documents internes, les communications à travers les vidéoconférences.

Les grandes organisations disposent aujourd'hui d'intranets dotés de serveurs et d'un grand nombre de postes de travail. Par exemple l'intranet du BRGM (Bureau de Recherches Géologiques et Minières) en France compte près de 2000 postes de travail. Dans certaines de ces

3. http ://www.top500.org
4. http ://www.top500.org/system/10184

organisations, les postes de travail sont très peu utilisés la nuit, les week-ends et pendant les périodes de congés, ce qui constitue une grande puissance de calcul libérée qui peut être exploitée pour faire du calcul haute performance.

Grappes

Une grappe (en anglais *cluster*) de calcul est l'interconnexion d'un ensemble homogène de nœuds indépendants à travers un réseau local rapide. Les communications entre les nœuds se font par échange de messages. Les nœuds sont des ordinateurs classiques contenant chacun un ou plusieurs processeurs. Les processeurs d'un nœud sont en général multicœurs, mais on en trouve encore qui sont monocœur. Le concept de grappe est né du projet *Beowulf* [19]. En effet en 1994 Thomas Sterling et *al* ont construit une grappe de 16 noeuds qui étaient des PCs standards constitués de processeurs Intel 80486 cadencés à 100 MHz, les noeuds étant interconnectés par un réseau 10 mégabits Ethernet. Pour exécuter leurs benchmarks, Sterling et *al* avaient installé sur cette grappe la bibliothèque de communication pour la programmation parallèle PVM (Parallel Virtual Machine). Les grappes au sens de *Beowulf* sont donc des environnements de calcul dédiés. Les grappes offrent un très bon rapport performance/coût et une grande capacité agrégée de mémoire qui font d'elles de sérieuses concurrentes des machines parallèles.

Grilles

Une grille (en anglais *Grid*) est un ensemble de grappes interconnectées entre elles par des réseaux de très haut débit (Figure 2.2). Plus généralement, une grille informatique est un système distribué composé du partage de ressources informatiques appartenant à plusieurs organisations. Elle offre aux utilisateurs une vue cohérente de ses ressources. Une grille est une architecture dédiée très hétérogène puisque les matériels et les systèmes qui la constituent peuvent être de différentes architectures. Une architecture générale et extensible des grilles a été présentée dans [20].

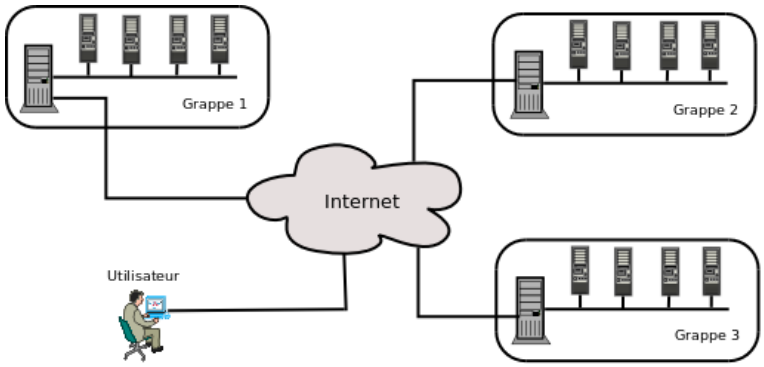

FIGURE 2.2 – Architecture générale des environnements de calcul de type grille

Les ressources d'une grille peuvent être déployées selon deux modèles : le modèle client/-

serveur et le modèle pair-à-pair. Dans le modèle client/serveur, les clients que sont les nœuds de la grille soumettent des requêtes à des serveurs qui centralisent les informations. Dans le modèle pair-à-pair, chaque noeud est à la fois client et serveur. La charge du réseau est répartie à travers les pairs qui sont responsables de la distribution de l'information.
Dans nos travaux nous nous intéressons aux grilles de calcul dont les ressources sont connectées suivant le modèle client/serveur.

C'est la grille française Grid5000 [21] qui a été notre environnement expérimental. Cette grille est constituée de grappes de calcul installées dans 9 villes (Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia Antipolis, Toulouse). Plus précisément les expériences présentées dans cette thèse ont été menées sur les sites de Bordeaux, Orsay et Sophia.

Depuis quelques temps, un nouveau concept est en pleine émergence : le *cloud computing* ou *nuage de calcul*. Il est assez difficile d'en donner une définition qui soit universelle [22]. Un nuage de calcul peut être vu comme un type de système parallèle et distribué constitué d'un ensemble d'ordinateurs interconnectés et virtualisés qui sont alloués de façon dynamique et présentés comme une ressource de calcul unifiée basée sur les accords de service établis par voie de négociation entre les prestataires de services et les consommateurs.

Internet

Internet est un système distribué à très grande échelle basé sur le modèle client/serveur constitué de ressources provenant d'ordinateurs de particuliers et de réseaux des grandes organisations. Ces ressources sont interconnectées à travers un réseau haut débit et communiquent à travers le protocole TCP/IP. Les ressources d'Internet sont présentées aux utilisateurs à travers une interface unique : le World Wide Web communément appelé Web.

L'agrégation des micro-périodes d'inactivité des millions de ressources d'Internet constitue une ressource considérable qui peut être utilisée pour déployer des applications de calcul haute performance. Ceci a suscité la création de projets qui exploitent les ressources d'Internet pendant leur période d'inactivité. Ces projets utilisent les ressources d'Internet pour reconstituer virtuellement les environnements de type grappes dédiées. Dans cette approche les propriétaires connectés à Internet offrent de manière explicite des portions inutilisées des ressources de leurs postes de travail. On parle alors de *Desktop Computing*. Les ressources libres telles que les temps de cycle du processeur, la mémoire et le disque sont collectées et intégrées à la grille ainsi formée.

Cette approche est utilisée dans le projet XtremWeb[23, 24] et dans les projets @home comme SETI@home [25], Folding@home [26], Einstein@Home [5] (qui effectue la détection de pulsars au moyen d'interféromètres laser), LHC@Home [6] (qui effectue des simulations de l'effet de particules à très haute vitesse sur l'accélérateur de particules du CERN : le LHC), Predictor@Home [7] (un projet qui tente de déterminer le repliement de protéines d'êtres vivants).

5. http ://einstein.phys.uwm.edu
6. http ://lhcathome.cern.ch/
7. http ://predictor.scripps.edu/

Nous présentons plus en détail les projets SETI@home et Folding@home et XtremWeb eu égard de la puissance de calcul qu'ils dégagent ou de leur particularité.

SETI@home

Le projet SETI@Home est basé sur l'agrégation de la puissance de calcul fournie par des millions de machines volontaires connectées à Internet.

SETI@home a pour objectif la détection de signaux radios émis par une intelligence extra-terrestre à partir des données observées par un télescope basé à Porto Rico. Ce projet consiste en une unique application de traitement des données observées. Sur chaque machine participante, un logiciel client libre est téléchargé et s'exécute en fond d'écran. SETI@home scrute chaque parcelle de l'espace et obtient des données qui sont stockées sur des bandes de 35 Go. Ces données sont ensuite divisées en fragments de taille de 250 Ko (chacun représentant 107 secondes d'observation). Le serveur de SETI@Home distribue les fragments aux machines d'Internet participant volontairement au projet.

Chaque machine volontaire analyse la zone reçue et le résultat de cette analyse est ensuite renvoyé au serveur qui se charge de fusionner et d'interpréter les résultats. Ce processus est itéré tant que la machine reste en mode écran de veille. Afin d'éviter que le calcul en cours ne soit perdu si l'utilisateur provoque la sortie en mode écran de veille, l'application SETI@home dispose d'un mécanisme qui va sauvegarder un point de reprise de l'application toutes les 10 minutes.

Folding@home

Le projet Folding@home a été conçu pour effectuer des simulations dans le domaine du repliement et l'agrégation des protéines. Folding@home se sert du mode écran de veille pour utiliser les périodes d'inactivité relativement courtes d'une machine donatrice. Il offre une grande rapidité de réaction au retour de l'utilisateur principal de la machine. La puissance de calcul dégagée par Folding@Home est proche du pétaflops. Folding@home a simulé un repliement de protéine pour la première fois avec une participation de 2 Millions de CPUs à travers le monde.

XtremWeb

XtremWeb est un de projet de recherche développé au Laboratoire de Recherche en Informatique (LRI), de Paris XI. XtremWeb exploite un modèle de déploiement pair-à-pair (P2P) centralisé et permet de virtualiser des groupes de ressources (PCs de volontaires connectés à Internet, stations de travail d'intranets d'entreprises, ressources provenant de grappes de calcul) au sein d'Internet, pour en faire une grille de calcul à grande échelle. Les participants à la plate-forme construite par XtremWeb coopèrent en fournissant le temps d'inactivité de leurs processeurs. L'architecture générale de XtremWeb repose sur trois composantes :

- le *client* qui permet de soumettre des tâches à la plate-forme et d'en récupérer les résultats de leur exécution. Le client est le point d'entrée à XtremWeb.
- les *travailleurs (workers)* sont les machines volontaires qui se proposent de faire des calculs durant leurs périodes d'inactivité. Un travailleur disponible contacte le serveur

pour récupérer des données, les traite et renvoie le résultat (jusqu'à 100 Mo) au serveur. Un travailleur peut aussi fonctionner en mode client et soumettre des travaux au serveur ;
– le *serveur (coordinator)*. Il reçoit les requêtes des clients pour les distribuer aux travailleurs qui demandent du travail. Il stocke aussi les résultats reçus des travailleurs pour les mettre à disposition des clients.

Les agents *client* et *worker* sont déployés sur une machine volontaire qui offre ses ressources à la plate-forme construite par XtremWeb. XtremWeb utilise un mécanisme de *sandboxing* [23] pour protéger les machines volontaires (travailleurs) contre toute attaque de sécurité qui pourrait provenir de codes malicieux.

Comme nous l'avons vu, les projets qui utilisent les ressources libres d'Internet tentent de reconstituer des environnements de calcul de type grille. Dans le cadre de cette thèse, nous nous proposons de faire de même mais à moyenne échelle au niveau des intranets, c'est-à-dire d'utiliser les ressources libres des intranets pour reconstituer virtuellement des environnements de calcul de type grappes. Cette idée est née du projet IGGI que nous présentons plus en détail dans la section suivante.

2.2 Projet IGGI

IGGI (Infrastructure pour Grappe, Grille et Intranet) [1] est un projet RNTL (Réseau National de recherche et d'innovation en Technologies Logicielles) qui a démarré en France en fin 2004 et s'est achevé en début 2007. Ce projet avait pour but de transformer l'intranet d'une organisation ou d'une entreprise en une infrastructure de calcul de type grappe virtuelle. Cette infrastructure devait servir de support pour les simulations et l'exécution des applications de calcul haute performance. Cette idée reste d'actualité et de nouveaux horizons sont actuellement envisagés pour la poursuite du projet.

Un bon nombre de défis que pose l'idée du projet IGGI reste à relever. Par exemple il serait intéressant de savoir comment gérer la libération des ressources de l'intranet lorsque les périodes d'indisponibilité tirent à leurs fins, alors qu'il y a encore des programmes en cours d'exécution. Il serait tout aussi intéressant de savoir comment utiliser efficacement les temps de jachère des ressources des intranets. Ces quelques questions ont été à l'origine de cette thèse.

Le projet IGGI était basé sur deux composantes principales : ComputeMode [27] chargé de basculer l'intranet en infrastructure de calcul et OAR [9] ou CIGRI [28] des gestionnaires de ressources.

2.2.1 OAR/CIGRI

OAR est un gestionnaire de ressources qui combine efficacité et passage à l'échelle pour manager les ressources d'une grappe que sont les noeuds de calcul. Nous reviendrons sur la présentation de OAR à la secttion 2.3.5.

CIGRI est un intergiciel qui permet à plusieurs utilisateurs de soumettre leurs travaux sur

des grappes de calcul. CIGRI est développé dans le laboratoire ID [8] de Grenoble au sein de l'équipe MESCAL. CIGRI est utilisé au BRGM comme gestionnaire de ressources tant dans la grappe dédiée que dans la grappe virtuelle déployée avec ComputeMode.

2.2.2 ComputeMode

Développé par la société ICATIS [9], ComputeMode est un logiciel libre qui permet de déployer temporairement Linux sur les PCs de bureau fonctionnant sous Windows à partir d'un serveur. Les postes passent ainsi du mode station de travail ou bureautique en mode ressource de calcul pour grappe virtuelle. ComputeMode offre des outils pour gérer simplement le cycle d'utilisation des machines. Il offre également des capacités d'administration à distance. Au terme de leur utilisation, Le serveur ComputeMode rebascule les ressources de la grappe en mode station de travail. ComputeMode est en voie de mise à jour afin qu'il soit opérationnel sur les plateformes actuelles.

La problématique de recherche se situant au niveau des architectures de type grappe, nous donnons dans la section suivante, une description des principales composantes d'une grappe.

2.3 Grappes de calcul

Du fait de son architecture homogène, une grappe de calcul s'apparente dans son fonctionnement à une machine parallèle. Dans cette section, nous présentons les composants de base qui sont utilisés pour construire une grappe. Ces composants sont les matériels ainsi que les outils logiciels qui permettent d'assimiler l'ensemble à une unique machine parallèle aux capacités décuplées pour un coût relativement bas. Pour plus de détail sur la construction des grappes, nous invitons le lecteur à se référer à l'ouvrage [29].

2.3.1 Architecure générale

La figure 2.3 présente une vue générale des environnements de calcul haute performance de type grappe. Les composantes de la grappe sont interconnectées à travers un réseau d'un haut débit en général. Les utilisateurs peuvent soumettre des travaux à travers le réseau Internet. Les requêtes sont transmises au gestionnaire de travaux et de ressources de la grappe via les frontales qui servent à authentifier les utilisateurs.

2.3.2 Nœuds de calcul

Les nœuds de calcul d'une grappe sont les PCs standards à faible coût que l'on trouve sur le commerce. Le choix de l'architecture des nœuds ainsi que le choix du réseau qui les

8. devenu LIG (Laboratoire d'Informatique de Grenoble)
9. http ://www.icatis.com

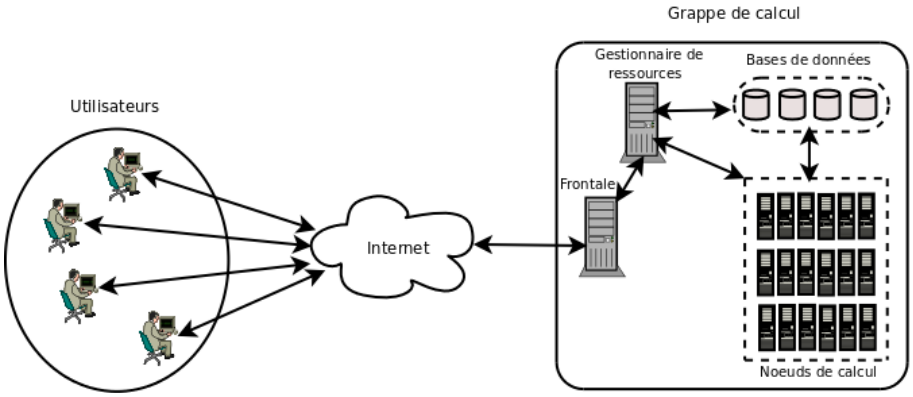

FIGURE 2.3 – Architecture générale des environnements de calcul de type grappe

interconnecte aura un impact déterminant sur les performances de la grappe. Cette section explore les principaux éléments matériels d'un nœud qui interviennent dans l'exécution et la sauvegarde des applications.

Processeurs

Le processeur (CPU : Central Processing Unit, en anglais) est le composant central de chaque ordinateur . Les calculs faits par un ordinateur sont exécutés par le processeur. Un nœud de calcul peut être doté de plus d'un processeur (monocœur ou multicœurs, mais la tendance aujourd'hui est multicœurs).

Un processeur exécute des instructions à une fréquence mesurée en mégahertz (MHz) ou gigahertz (GHz). La fréquence est utilisée pour classer les processeurs d'une même famille. Par exemple pour les processeurs d'Intel, on parlera de Pentium III 750Mhz (1999), Pentium 4 1,7GHz (2001), Dual core 2GHz (2007). Elle ne permet pas cependant de comparer les processeurs de familles différentes (par exemple un Intel et un Opteron).

Un autre paramètre déterminant dans les processeurs c'est le format du mot machine et de l'adressage. De nos jours, l'architecture 64 bits (IA-64, AMD64 [10] , IBM Power, Sun Sparc64 [11], MIPS, Alpha) permet de gérer des espaces d'adressage plus importants.

Mémoire et cache

La mémoire (RAM : Random Access Memory) est une zone de stockage temporaire des instructions et des données. Le processeur fait constamment des accès à la mémoire pour stocker ou prendre des informations à travers le bus mémoire. Pour des applications très soigneusement conçues, les données d'un programme devraient résider entièrement dans la mémoire. Les algorithmes *out-of-core* font référence aux applications dont le volume de données ne peut pas

10. http ://www.amd.com/us-en/Processors/ProductInformation/
11. http ://www.sun.com/processors/

tenir en mémoire. Ces algorithmes [30] sont optimisés pour accéder de manière explicite aux données stockées dans le disque dur sans utiliser les fonctionnalités de *swap* du système d'exploitation. La fonction de *swap* est directement intégrée dans l'algorithme. Cela implique en général de sévères pénalités sur la performance des applications et du système. Ainsi, la taille de la mémoire d'un nœud est un paramètre important dans la conception de la grappe. Elle détermine la taille du problème qui peut pratiquement être exécuté sur le nœud.

L'accès mémoire est beaucoup plus lent que la vitesse du processeur. Le processeur dispose de plusieurs niveaux de mémoire cache d'accès plus rapide que la mémoire, pour stocker les blocs de mémoire auxquels on a régulièrement accès. Les architectures actuelles possèdent une hiérarchie de mémoire importante : le cache L1, le plus petit, est utilisé pour contenir une partition des données et des instructions du programme. Les caches L2 et L3 sont généralement utilisés pour les données du programme.

Disque

Les disques sont des mémoires non volatiles qui permettent de stocker de façon permanente les applications (séquentielles ou parallèles) ainsi que les données et résultats. Ils peuvent aussi servir de mémoire secondaire pour les applications nécessitant de stocker de très gros volumes de données au cours de leur exécution. La capacité et la vitesse d'accès au disque augmentent rapidement d'année en année.

Les disques sont accessibles à travers des interfaces de bus. Les bus les plus couramment utilisés pour les disques standards sont : IDE (ou ATA), SCSI et SATA (Serial ATA). Les bus SATA ont été conçus pour apporter de meilleures performances que les IDE/ATA. Les bus SCSI offrent des débits intéressants, mais leur coût est plus élevé, comparativement à ceux des bus IDE.

Carte réseau

Les cartes réseaux sont des composants qui servent de points d'entrée/sortie d'information au niveau de chaque nœud d'une grappe. Les nœuds peuvent avoir une ou plusieurs cartes réseaux à travers lesquelles ils envoient et reçoivent des messages sur le réseau.

Les cartes réseaux disposent de composants matériels et logiciels pour effectuer les opérations d'entrée/sortie. Lorsqu'une application s'exécutant sur un nœud veut transmettre un message vers le réseau, l'information transite à travers le bus PCI (ou PCI-X pour certaines architectures) et est d'abord copiée dans la mémoire tampon (*buffer*) de la carte réseau en vue d'être préparée à la transmission. Un processeur spécifique contenu dans la carte réseau exécute alors un protocole qui convertit le message à transmettre en un message au format du réseau avant la transmission. Ce protocole effectue l'opération inverse pour les messages provenant du réseau et qui doivent transiter à travers le bus PCI vers leur destination sur le nœud.

2.3.3 Réseaux d'interconnexion

Dans une grappe les nœuds communiquent à travers des protocoles de communication. Ces communications sont réalisées par des réseaux dont les performances dépendent généralement de trois métriques : la *latence*, la *bande passante*, la *topologie* du réseau. La latence est le temps de transit d'un message du nœud émetteur vers le nœud récepteur. La bande passante est la vitesse de transfert des messages. La topologie du réseau est l'organisation sous-jacente du réseau. Les réseaux opèrent à travers un certain nombre de composants : les *cartes réseaux* qui sont connectées au réseau à travers les *liens réseaux* et les *commutateurs*.

Cartes réseaux

Les cartes réseaux ont été présentées à la section 2.3.2. Mais il convient de préciser que l'architecture de la carte réseau est dépendante du type de réseau utilisé. Par exemple les cartes réseaux du réseau *Gigabit Ethernet* ont une mémoire tampon de 96 Ko alors que celles du réseau *Myrinet* ont une mémoire tempon dont la taille varie entre 2 Mo et 4 Mo.

Liens réseaux

Les liens réseaux sont des canaux connectant les cartes réseaux aux commutateurs. La vitesse des liens varie en fonction des technologies réseaux. : 10 Mb/s (Mb : Mégabit), 100 Mb/s, 1 Gb/s. Un lien peut être half-duplex ou full-duplex. Pour un lien half-duplex, des transmissions simultanées sur le lien peuvent causer des collisions. Ces collisions peuvent être des causes de perte de performance du réseau. En effet la bande passante disponible pour tous les nœuds qui transmettent simultanément vers une même destination est celle de l'unique lien sollicité. En générale la bande passante théorique d'un lien est la bande passante théorique du réseau.

Commutateurs

Les commutateurs sont des composants qui interconnectent les nœuds à travers des liens réseaux. Ils utilisent des algorithmes spécifiques pour déterminer les chemins que doivent emprunter les paquets. Les commutateurs disposent d'un certain nombre de ports qui indiquent le nombre de nœuds qui peuvent être connectés à la fois. Ce nombre indique ainsi la bande passante agrégée des liens et par conséquent le volume de messages que le commutateur peut manipuler à la fois.

Réseaux haut débit

Les réseaux haut débit usuels dans les grappes sont : *Gigabit Ethernet*, *Myrinet*, *Quadrics* et *InfiniBand*. Les nœuds des grappes que nous avons utilisées étaient connectés par un réseau *Gigabit Ethernet* que nous présentons dans le paragraphe suivant. Le lecteur peut se référer à [31] pour les informations sur le réseau *Myrinet*, [32] pour le réseau *Quadrics* et [33] pour le

réseau *InfiniBand*.

Ethernet

Le standard *Ethernet* (10 Mb/s) a été et est encore largement utilisé dans les réseaux de stations de travail. Le calcul haute performance sur les grappes induit généralement une grande activité de transfert dans le réseau, dû à la grande quantité de données acheminées entre les nœuds. Le débit de 10 Mb/s du standard *Ethernet* est alors une limitation à la performance de la puissance de calcul de la grappe. *Fast Ethernet* est une amélioration du standard *Ethernet* qui offre un débit de 100 Mb/s. Le succès de *Fast Ethernet* dans les grappes a été interrompu avec la construction de son successeur *Gigabit Ethernet* qui offre une bande passante avec un pic de 1 Gb/s. *Gigabit Ethernet* tend à être remplacé avec l'émergence de la technologie *10 Gigabit Ethernet* qui utilise des liens full-duplex.

2.3.4 Système de fichier réseau

Dans la plupart des grappes de calcul, la sauvegarde des données est généralement effectuée sur les serveurs de fichiers. Ces serveurs peuvent également être utilisés pour le partage des données à travers des systèmes de fichiers partagés. L'accès aux fichiers pour les opérations de lecture/écriture par un nœud est réalisé de façon transparente par un protocole client-serveur standard.

Le protocole NFS (Network File System) [34, 35] est le plus largement utilisé parmi les propositions existantes, dans les grappes sous Linux. NFS permet d'exporter une partition du disque du serveur de fichiers vers chaque nœud de la grappe. Les nœuds, peuvent alors accéder à cette partition comme un répertoire local. Pour réaliser cela, NFS utilise les protocoles VFS (Virtual File System), RPC (Remote Procedure Call) et XDR (External Data Representation) introduits par SUN. L'une des limitations de NFS est le passage à l'échelle. Les performances de NFS se dégradent pour des grappes ayant plus de 64 nœuds. NFS est très indiqué pour les grappes de petite taille.

Pour les grappes de grande taille et ayant plusieurs serveurs, les systèmes de fichier réseau tels que GPFS [36] et PFVS [37] sont souvent utilisés.

2.3.5 Gestionnaires de ressources

Le gestionnaire de ressources (*batch scheduler*), aussi appelé gestionnaire de travaux, est un élément central parmi les outils nécessaires pour l'exploitation d'une grappe de calcul. L'objectif d'un gestionnaire de ressources est de permettre aux usagers d'utiliser facilement les ressources (les nœuds) de la grappe. Pour atteindre cet objectif, un gestionnaire de ressources exécute en général un certain nombre d'activités parmi lesquelles :

– la gestion des files d'attente des travaux. Le gestionnaire enregistre les soumissions des travaux des utilisateurs dans une file d'attente avant de les exécuter suivant une politique bien définie, en fonction des ressources disponibles. La soumission d'une tâche est décrite

par exemple par le nombre de nœuds (ou de processeurs ou de cœurs) dont a besoin un utilisateur, le nom du programme à exécuter et la liste des arguments en entrée du programme, la liste des données nécessaires à l'exécution de la tâche.
– l'ordonnancement des exécutions. Cela consiste pour un gestionnaire de travaux à déterminer quelles tâches doivent être exécutées sur quels nœuds et à quel moment. La politique d'ordonnancement devrait permettre une gestion optimisée des ressources de la grappe.
– le monitoring. C'est une activité du gestionnaire de ressources qui fournit aux administrateurs de la grappe, aux utilisateurs et au gestionnaire lui-même, des informations nécessaires sur le statut des tâches et des ressources.
– la gestion des ressources. Cette activité consiste à ajouter ou à retirer des ressources de la grappe. Elle consiste également à arrêter une tâche et à libérer proprement les ressources lorsqu'une tâche est achevée ou arrêtée par le gestionnaire de travaux.

OAR

Les grappes de Grid5000 sont administrées par le gestionnaire de ressources *OAR* [9]. *OAR* est un produit du laboratoire ID-IMAG [12]. Il est développé en langage de script *Perl* et repose sur le gestionnaire de base de données Mysql pour une extraction efficace des données. *OAR* est composé de modules qui interagissent seulement avec la base de données et sont exécutés comme des programmes indépendants. OAR gère les tâches avec des files où chaque file a une priorité et chaque file a sa propre politique d'ordonnancement. *OAR* accorde une forte priorité aux tâches soumises en *Best effort* (celles qui peuvent s'achever avant la fin du temps qui leur a été alloué). Tout ceci fait de *OAR* un gestionnaire de travaux adapté pour une gestion efficace des ressources des grappes de Grid5000.

Cependant, jusqu'au moment de la rédaction de ce manuscrit, aucune version de *OAR* ne prend en compte la sauvegarde des applications. *OAR* offre la possibilité à l'utilisateur de fournir un délai, lors de la soumission d'une tâche, qui sera utilisé pour prévenir cet utilisateur de l'éminence de la fin du temps qui lui est alloué. L'utilisateur devra alors introduire explicitement dans son code, des mécanismes pour capter le signal envoyé par OAR pour effectuer lui-même la sauvegarde de son application avant la fin du temps qui lui est alloué.

Autres gestionnaires de travaux

Il existe un grand nombre de gestionnaire de travaux pour les grappes et les machines parallèles. Nous pouvons par exemple citer parmi les plus connus : PBS/OpenPBS [38, 39], Condor [5], Maui [40] et SGE [41] devenu N1GE6 [42] depuis 2004. Certains de ces gestionnaires de travaux comme Condor et N1GE6 intègrent la prise en charge de la sauvegarde automatique des applications avec une périodicité qui peut être soit par défaut, soit fixée par l'utilisateur.

Cette section a présenté les environnements de grappes ainsi que leurs principales composantes qui en font des environnements de calcul haute performance. Dans la section qui suit,

12. devenu LIG (Laboratoire d'Informatique de Grenoble)

nous présentons les différentes approches d'utilisation des ressources libres des intranets pour faire du calcul haute performance.

2.4 Grappes virtuelles

Différentes approches sont utilisées pour agréger les ressources libres provenant d'intranets pour en faire des grappes de calcul haute performance. La plate-forme ainsi obtenue porte le nom de *grappe virtuelle*. Une étude plus approfondie sur l'agrégation des ressources inutilisées d'un intranet en vue de créer une infrastructure de calcul a été menée dans [43] dans le cadre du projet *I-Cluster*.

2.4.1 Utilisation directe des ressources disponibles

Dans cette approche les propriétaires offrent de manière explicite des portions inutilisées des ressources de leurs postes de travail pour la grappe qui les sollicite. Les ressources libres telles que les temps de cycle du processeur, la mémoire et le disque sont collectées et intégrées à la grappe. La grappe virtuelle ainsi constituée peut disposer d'une puissance de calcul considérable. Lorsqu'une machine donatrice est inactive, un *agent* est démarré sous le système d'exploitation natif (Figure 2.4) pour l'exécution d'un calcul scientifique. La contrainte dans cette approche est que les applications doivent être exécutées avec les bibliothèques du système d'exploitation natif. Ceci peut engendrer un importante activité de sauvegarde locale et une interférence avec l'utilisateur principal de la machine.

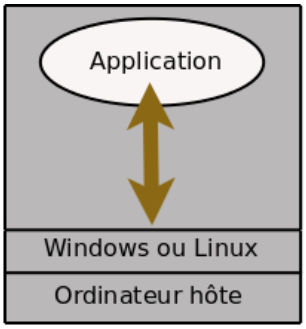

FIGURE 2.4 – Calcul dans l'environnement de travail

Cette approche est utilisée dans le projet Condor [5, 44].

Condor

Le label *Condor* fait allusion à la fois à un projet de recherche, à un gestionnaire de ressources et à un système de sauvegarde d'applications de calcul haute performance.

Le projet Condor utilise un système de publication dans lequel chaque machine participante expose ses capacités de calcul et ses ressources disponibles à un serveur. Ces ressources sont

utilisées pour faire du calcul haute performance. Les travaux soumis à Condor sont distribués aux stations de travail distantes par le gestionnaire de travaux *Condor Manager*, en faisant une utilisation efficace des ressources réseau disponibles. Lorsqu'un utilisateur soumet une tâche, Condor va trouver la machine disponible la plus appropriée pour l'exécution du travail.

Une contrainte inhérente à l'utilisation de Condor est que les travaux à exécuter doivent être recompilés avec des modules de Condor.

L'exécution des calculs dans le même environnement que celui de l'utilisateur pose des problèmes d'interférence avec l'utilisateur principal de la machine. Une autre approche consiste à utiliser des machines virtuelles.

2.4.2 Exécution dans des machines virtuelles

La technologie de *machine virtuelle* permet d'exécuter plusieurs systèmes d'exploitation sur une même machine physique (Figure 2.5). Les systèmes étrangers et le système natif se partagent les ressources de la machine hôte. Cette approche réduit les interférences au niveau du disque local avec l'utilisateur principal.

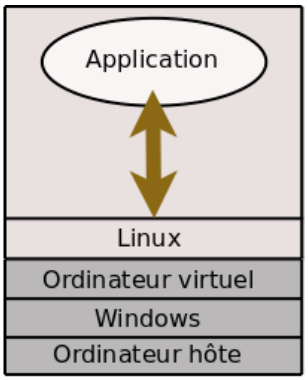

FIGURE 2.5 – Calcul dans une machine virtuelle

L'avènement des machines virtuelles a suscité un grand intérêt pour les environnements de grappes et de grilles de calcul. En général, les ressources offertes par les grappes et les grilles de calcul ne satisfont pas toujours aux besoins des utilisateurs qui veulent exécuter leurs applications ou satisfaire à des services. Ceci peut résulter en une déception de la communauté des utilisateurs et une sous-utilisation des ressources (puisque celles allouées ne satisfont pas à la demande). Pour résoudre ce problème l'utilisation des machines virtuelles a été proposée [45, 46]. Le projet In-Vigo [47] propose l'utilisation d'une infrastructure de calcul distribué basée sur les machines virtuelles. Le projet Violin [48] explore l'utilisation des machines virtuelles pour une bonne gestion des ressources partagées des infratructures de calcul distribué. Dans [49], Ian Foster et *al.* proposent une description des protocoles permettant à un utilisateur d'une grille de déployer une grappe virtuelle appropriée à ses besoins.

Les logiciels libres VmWare [13], VirtualBox [14] et Xen [50] utilisent les moniteurs de machine virtuelle (VMMs) [51] et fournissent une abstraction permettant d'exécuter plusieurs machines virtuelles sur une machine physique.

La technologie de machine virtuelle a fait de très grands progrès et les performances des machines virtuelles approchent celles de la machine hôte [52]. Ceci motive de plus en plus l'utilisation des machines virtuelles pour le calcul haute performance dans les environnement de grappes. Cependant l'exécution d'applications scientifiques sur les machines virtuelles peut avoir un impact sévère sur la performance des applications [53]. En effet, cette perte de performance provient de ce que les applications, aussi bien que le système d'exploitation virtuel et le système natif, se partagent la mémoire de la machine physique. Cette situation ralentit encore l'utilisation à grande échelle des machines virtuelles pour le calcul haute performance dans les environnements de grappes.

Les approches présentées dans les sections 2.4.1 et 2.4.2 sont intéressantes pour les environnements où les postes de travail sont au repos pendant de courtes périodes (de l'ordre de quelques minutes). Elles offrent une bonne rapidité de réaction au retour de l'utilisateur principal. Lorsque les postes sont disponibles pour de longues périodes (nuits, week-ends, périodes de congé), une bonne approche est de les basculer en mode de machine de calcul dans un environnement dynamique de calcul. Cette approche est présenté dans la section suivante.

2.4.3 Bascule en mode grappe de calcul

Lorsque les ressources d'un intranet sont libres pour une longue période de temps, il est intéressant de les rebooter pour les intégrer à une grappe de calcul virtuelle. Dans cette approche, les postes de travail retenus pour le cluster virtuel, sous les systèmes Windows ou linux, sont rebootés sous Linux (Figure 2.6) avec un outil tel que ComputeMode [27], tout comme l'avait envisagé le projet IGGI [1].

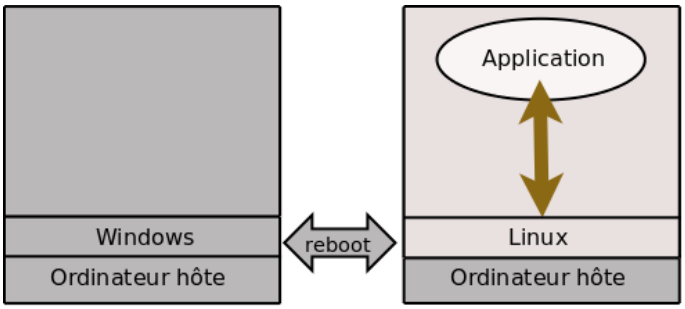

FIGURE 2.6 – Calcul dans une machine en mode *sans disque*

Les postes acquis sont démarrés en mode *sans disque*, par conséquent les seules ressources utilisées sont le CPU et la RAM. Avec cette approche, les opérations de lecture/écriture des

13. http ://www.vmware.com
14. http ://www.virtualbox.org

données sont effectuées au niveau du serveur de l'intranet. Ceci permet d'éviter toute panne qui pourrait subvenir suite à l'utilisation d'un disque local. Cette approche présente cependant une faiblesse : le serveur peut devenir un goulot d'étranglement lorsque plusieurs postes de travail y font accès simultanément en écriture.

Cette dernière approche a été retenue dans le cadre des travaux présentés dans cette thèse. Elle offre plus de disponibilité pour l'utilisation des ressources car les travaux en cours d'exécution sur les postes acquis ne sont pas régulièrement perturbés.

2.5 Conclusion

Dans ce chapitre nous avons présenté les environnements de calcul de type grappe et les environnements fortement dynamiques de type grappe virtuelle. Nous avons précisément présenté les différentes composantes qui constituent les grappes et leur rôle dans les performances de telles infrastructures.

Ce chapitre a également abordé les différentes approches d'exploitation des ressources libres pour en faire des environnements de calcul haute performance tels que les grappes virtuelles. De l'analyse que nous avons menée, il ressort que l'approche d'exécution directe des calculs scientifiques dans l'espace utilisateur sur les machines disponibles et l'approche d'exécution dans des machines virtuelles tournant sur les machines volontaires présentent des problèmes d'interférence avec les utilisateurs principaux des machines disponibles. Il ressort également que ces deux approches sont indiquées pour exploiter les périodes de micro-inactivité des machines.

Nous avons présenté une autre approche qui consiste à n'utiliser que les machines inactives pour de longues périodes d'un intranet. Dans cette approche les postes acquis sont basculés en mode machine de calcul et intégrés à une grappe virtuelle. Pour éviter toute interférence avec l'espace utilisateur, seules les ressources comme le processeur et la mémoire sont utilisées sur les postes acquis. Cette approche a été adoptée dans le cadre de cette thèse. Etant donné l'état *diskless* (sans disque) des machines acquises pour la grappe virtuelle, toutes les opérations d'entrée/sortie se font sur le disque du serveur. Dans ce contexte, le serveur peut devenir un goulot d'étranglement lorsque plusieurs applications s'exécutant sur différentes machines de la grappe font des accès simultanés en écriture vers le serveur. Nous étudierons ce problème au chapitre 4.

La volatilité des ressources dans les environnements de grappe virtuelle ne permet pas toujours d'exécuter complètement les applications. Si aucune disposition n'est prise, on perdrait un grand volume de calcul effectué pour une application inachevée, au moment de la libération des ressources de la grappe virtuelle. La sauvegarde des applications en vue de leur reprise lorsque les ressources sont à nouveau disponibles permet de pallier ce problème.

Dans le chapitre suivant nous allons présenter les mécanismes et les différentes approches de sauvegarde/reprise d'applications de calcul haute performance dans les environnements dynamiques.

Chapitre 3

Mécanismes de sauvegarde/reprise d'applications

Sommaire

Dans le chapitre précédent, nous avons montré les différentes approches d'utilisation des ressources libres des intranets au cours des périodes de micro-inactivité et de longues périodes d'inactivité comme les nuits, week-ends et jours fériés, pour en faire des environnements de calcul haute performance telles que les grappes virtuelles. Généralement, ces périodes d'inactivité ne permettent pas toujours d'exécuter les calculs scientifiques jusqu'à leur terme. Il est donc nécessaire de sauvegarder le contexte d'exécution des applications pour d'éventuelles reprises. Dans ce cas les mécanismes de sauvegarde/reprise d'applications sont une bonne solution.

Dans ce chapitre nous présentons les mécanismes de sauvegarde/reprise d'applications. Nous discutons sur les différentes approches d'utilisation de ces mécanismes pour sauvegarder le contexte d'exécution des applications dans les environnements dynamiques. Nous présentons les problèmes posés pour la sauvegarde en parallèle d'applications dans le contexte où les ressources libres d'un intranet sont rebootées et intégrées dans une grappe virtuelle comme machine de calcul. L'approche de grappe virtuelle que nous avons adoptée comme cadre de travail dans cette thèse est un environnement fortement dynamique dans lequel les seules ressources utilisées d'un noeud sont la RAM et le CPU.

3.1 Définition et intérêt de la sauvegarde/reprise

Dans cette section nous présentons le concept de sauvegarde/reprise d'application et nous montrons son utilité fondamentale dans les environnements à mémoire distribuée.

3.1.1 Définition

La *sauvegarde/reprise* (*checkpointing* en anglais) est le processus qui consiste à sauvegarder le contexte d'une application en cours d'exécution sur un support stable. Le fichier ainsi sauvegardé, aussi appelé *point de reprise* (*checkpoint* en anglais), contient toutes les informations nécessaires à la reprise de l'application.

Dans la suite nous utiliserons indifféremment les termes sauvegarde/reprise et checkpointing.

3.1.2 Intérêt du checkpointing

L'objectif fondamental du checkpointing est d'établir des points de reprise pour une application en cours d'exécution et de sauvegarder un grand nombre d'images de l'application en vue d'une reprise en cas de panne. Ceci permet d'assurer un certain nombre de fonctionnalités dans les systèmes distribués, parmi lesquelles : la migration de processus, la tolérance aux fautes, la préemption.

Migration de processus

La migration de processus permet de déplacer une tâche d'une machine vers une autre. Cette opération vient en complément à la sauvegarde des points de reprise. La migration de processus est particulièrement utilisée en cas de crash d'un nœud. Elle est également utilisée pour équilibrer les charges d'un système. Par exemple en cas de surcharge d'un nœud, certains processus s'exécutant sur ce nœud peuvent être préemptés et migrés vers un nœud dont les ressources sont disponibles.

Tolérance aux fautes

Laprie et al. ont défini dans [54] certains concepts fondamentaux comme une *défaillance*, une *erreur* ou une *faute*. Une *défaillance* d'un système est définie comme un dysfonctionnement du système, c'est-à-dire que le système ne réagit plus conformément à ses spécifications. Une *panne* (ou *erreu*r) est un état d'un système susceptible de générer une défaillance. Une *faute* est la cause d'une erreur.

Une des caractéristiques qui distingue les systèmes distribués des machines parallèles est la notion de *défaillance partielle*. Dans les systèmes distribués, on parle de défaillance partielle lorsqu'une panne survient sur l'une des composantes du système. Á l'opposée, une défaillance dans les systèmes centralisés est toujours totale, en ce sens qu'elle peut entraîner l'arrêt total de l'exécution d'une application sur le système.

La tolérance aux fautes est la capacité pour un système de se rétablir automatiquement et de façon transparente après une panne. En d'autres termes, un système est considéré comme tolérant aux fautes si, lorsqu'une panne survient sur une de ses composantes, il continue de fonctionner de façon acceptable pendant que les réparations se font parallèlement.

La tolérance aux fautes est particulièrement intéressante pour des applications qui nécessitent de longues périodes d'exécution (des heures voire des jours). Elle est réalisée au moyen de sauvegardes périodiques du contexte d'exécution des applications. Ainsi, en cas de défaillance, l'application est reprise à partir de la dernière sauvegarde effectuée, ce qui permet de ne pas reprendre l'exécution depuis le début.

La réalisation de la tolérance aux fautes se fait en quatre phases : la détection des erreurs, le recensement des dommages, la correction des erreurs et la reprise sur panne. La détection des erreurs est la phase où la présence d'une faute est déduite à partir d'une panne survenue sur une composante du système. Une fois la panne détectée, cela implique une défaillance de la composante. Tout dommage dû au dysfonctionnement (défaillance) est identifié et circonscrit dans la phase de recensement des dommages. Après ces deux phases, la panne doit être corrigée et ceci se fait dans la phase de correction des erreurs. La quatrième phase, celle de la reprise sur panne, devra permettre au système de fonctionner soit sans la composante où la panne est survenue, ou alors avec cette composante mais reconfigurée de sorte que la panne ne puisse plus causer de défaillance. Dans [55], Pankaj Jalote aborde plus amplement la question de la tolérance aux fautes dans les systèmes distribués.

La tolérance aux fautes a été étudiée pour faire face aux pannes qui peuvent arriver dans

le système de façon imprévisible. Dans le cadre de nos études où les ressources libres d'un intranet sont utilisées comme ressource de calcul et doivent être restituées, l'on peut considérer la date de restitution des ressources comme date d'une panne générale qui a entraîné l'arrêt des machines. Nous nous situons alors dans le cas où la panne est connue. Ceci nous amène à aborder différemment le problème de la sauvegarde/reprise, comme nous allons le voir dans la suite de ce chapitre.

Préemption

La préemption consiste à stopper temporairement une tâche dans le but d'allouer les ressources qu'elle utilisait à une autre tâche. La préemption consiste aussi à stopper momentanément un groupe de processus ou l'ensemble des processus d'une grappe pour des raisons de maintenance en cas de défaillance sur un groupe de nœuds ou sur toute la grappe. La préemption est nécessaire pour implémenter les notions d'ordonnancement (telle que la priorité) et le débogage d'applications.

Notre intérêt pour le checkpointing se situe au niveau de la préemption. En effet, lorsque l'on sauvegarde le contexte d'exécution des applications dans des environnements dynamiques pour d'éventuelles reprises, on n'est pas sûr de pouvoir disposer des mêmes postes de travail lors de la prochaine disponibilité des ressources de l'intranet. Les reprises peuvent alors être effectuées sur de nouvelles machines. Tout se passe alors comme si les processus ont été préemptés pour des raisons de maintenance.

Le concept de sauvegarde/reprise étant bien défini et son intérêt présenté, nous pouvons dès lors nous intéresser aux techniques qui contribuent à leur mise en place. La section qui suit apporte des informations à propos.

3.2 Checkpointing d'applications

La complexité des mécanismes de sauvegarde/reprise diffère suivant deux classes d'applications : les applications séquentielles et les applications parallèles.

3.2.1 Checkpointing d'applications séquentielles

La sauvegarde d'une application séquentielle en cours d'exécution consiste en la sauvegarde de tout ou d'une partie du contexte d'exécution de son processus. Un processus UNIX est constitué d'un espace d'adressage mémoire et d'autres informations à propos du processus gérées par le noyau telles que l'état des registres du processus et les descripteurs de fichiers ouverts par le processus. L'espace d'adressage est généralement divisé en zone de texte contenant le code de l'application, la zone de données contenant les variables globales et les données allouées, et la pile. Du contexte d'un processus, seuls la zone de données, la pile et les registres nécessitent d'être sauvegardés. En général le processus à sauvegarder est momentanément interrompu lors

de l'initialisation d'une opération de sauvegarde/reprise. Plus de détails techniques sont fournis dans [56, 57, 58].

3.2.2 Checkpointing d'applications parallèles

Dans un système mono-processus, il est plus facile de reprendre l'exécution d'une application après la survenue d'une panne. Il suffit de restaurer le dernier checkpoint sauvegardé de l'application, étant donné que toutes les informations requises pour la reprise se trouvent dans ce fichier.

Dans les systèmes distribués, où les applications sont constituées de plusieurs processus qui communiquent par échange de messages, la sauvegarde et la reprise d'une application n'est plus aussi simple, étant entendu que l'état du système dépend maintenant des états des différents processus s'exécutant sur différents nœuds et des messages dans les canaux de communication. Chaque nœud peut établir un checkpoint localement (c'est-à-dire sur ses données seulement) à un instant donné. Cependant il n'existe pas de méthode générale pour établir un checkpoint au même moment sur tous les nœuds en vue d'obtenir un *état global consistant* du système à un instant donné. Un *état global consistant* est un état d'un système distribué à partir duquel on peut reprendre une application comme s'il s'agissait d'un système mono-processeur. Dans un état global consistant, si l'état local d'un processus P indique qu'il y a eut accusé de réception d'un message m d'un processus Q, alors l'état local de Q doit indiquer que le message m a été envoyé à P [2]. Dans ce contexte, on considère un checkpoint global consistant comme étant un ensemble de checkpoints locaux formant un état global consistant.

Dans le contexte d'applications parallèles, on dénote deux principales techniques : la sauvegarde/reprise non coordonnée et la sauvegarde/reprise coordonnée.

Sauvegarde/reprise non coordonnée

Dans l'approche non coordonnée, les processus déterminent localement leurs checkpoints de façon indépendante. Dans la phase de reprise, les processus recherchent un ensemble de checkpoints pour former un état consistant à partir duquel ils pourront reprendre leurs exécutions. Au cours de la recherche d'un ensemble de points de reprise consistant, il peut y avoir des ensembles de checkpoints dans lesquels on retrouve les situations suivantes :

- l'état sauvegardé d'un processus P indique qu'il a envoyé un message m au processus Q, mais l'état sauvegardé de Q n'indique pas d'accusé de réception du message m. Ce scénario présente un *message perdu* ;
- l'état sauvegardé de Q est tel qu'il indique un accusé de réception d'un message m de P, alors que l'état sauvegardé de P ne mentionne aucun envoi d'un message m à Q. Un tel message m qui a été reçu sans avoir été envoyé est appelé *message orphelin.*

Pour éviter la perte de message et la présence de message orphelin durant les retours arrière (*roll-back*) dans la recherche d'un état global consistant, des restrictions doivent être imposées à la collection de checkpoints des différents processus en vue de former un état consistant tolérant

aux fautes [55] :

- l'ensemble des points de reprise contient exactement un checkpoint pour chaque processus ;
- il n'existe pas d'événement d'envoi de message d'un processus P qui succède le checkpoint de P, dont l'événement correspondant d'accusé de réception d'un processus Q survient avant le checkpoint de Q. Ceci signifie en fait qu'il n'existe pas de message orphelin ;
- il n'existe pas d'événement d'envoi de message d'un processus P qui précède le checkpoint de P, dont l'événement correspondant d'accusé de réception d'un processus Q survient après le checkpoint de Q. Ceci signifie en fait qu'il n'existe pas de message perdu.

L'ensemble des points de reprise satisfaisant ces trois conditions est appelé *ligne de recouvrement*

Bien que l'approche asynchrone soit moins complexe (puisqu'il n'y a pas de coordination entre processus), elle présente quelques désavantages. Tout d'abord la possibilité que l'*effet domino* [55] se produise, ce qui peut ramener le système au début de l'exécution de l'application dans la recherche d'un état consistant. Par ailleurs un processus peut créer des checkpoints qui pourraient ne pas faire partie d'un état consistant. Enfin, puisqu'un état qui est bien en arrière peut être restauré, plusieurs points de reprise peuvent nécessiter d'être sauvegardés, ce qui accroît le volume de données à sauvegarder. En somme la reprise après une panne a un coût très élevé dans l'approche asynchrone.

Plusieurs modèles discutant de la détermination d'un état global consistant ont été proposés [59, 60, 61].
Juang et Venkatesan [61] ont proposé un algorithme itératif tolérant aux fautes, qui se résume ainsi : en cas de reprise, un processus P_i fait un roll-back sur ses précédents checkpoints jusqu'à ce qu'il soit sur un checkpoint dans lequel le nombre de messages reçus d'un autre processus P_j soit inférieur au nombre de messages envoyés par P_j, enregistrés dans le checkpoint actuel de P_j. De ce fait, après une série de roll-back, l'état de P_i est en cohérence parfaite avec l'état courant de ses voisins. Informé de cette situation, un processus P_j peut être amené à faire des roll-back pour rechercher aussi un état cohérent avec ses voisins. Le roll-back de P_j peut annuler l'action de P_i au point d'amener P_i à faire d'autres itérations dans la recherche d'un nouveau checkpoint cohérent avec ses voisins.
Juang et Venkatesan ont montré dans [62] qu'après N itérations de l'algorithme, N étant le nombre de processus du système, le checkpoint de chaque processus est cohérent avec ceux de ses voisins, entraînant ainsi l'obtention d'un état global consistant.

Dans [63] une approche de journalisation des messages est proposée pour éviter les pertes de messages et les messages orphelins, permettant ainsi de garantir une reprise cohérente du système. Dans cette approche, un message est envoyé à tous les processus, indiquant le niveau de lecture des tampons des sockets de l'application surveillée. A la réception de ce message, chaque processus met à jour ses journaux en supprimant les messages lus par l'application au moment du checkpointing.

L'approche de sauvegarde/reprise non coordonnée induit la sauvegarde d'un grand nombre de points de reprise. Cette approche a été conçue lorsque le dispositif des communications

réseaux présentait un coût plus élevé que les supports de sauvegarde. Avec la montée des réseaux à très haut débits, la tendance s'est inversée et cette approche est de moins en moins utilisée.

Sauvegarde/reprise coordonnée

Dans l'approche coordonnée, les processus dans le système coopèrent pour établir localement leurs checkpoints, de sorte que l'ensemble des checkpoints résultant forme un état consistant. Dans le checkpointing coordonné, la reprise de processus est simplifiée et il n'y a pas de possibilité que l'effet domino se produise. De plus cette approche minimise le volume de données à sauvegarder, du fait qu'un seul checkpoint permanent nécessite d'être sauvegardé sur un support de masse. Cette approche est plus complexe car, étant donné que les processus coopèrent entre eux, il faudra tenir compte des messages dans les canaux de communication au moment de la sauvegarde du contexte global.

Chandy et Lamport ont proposé dans [2] une approche pour le checkpointing coordonné. Dans cette approche, un processus initiateur lance la procédure de recherche d'un point de reprise global, et les processus coopèrent pour enregistrer localement leurs checkpoints, et coopèrent également pour enregistrer l'état des canaux de communication en vue d'obtenir un état global consistant.

Chandy et Lamport ont certes prouvé l'efficacité de leur modèle dans la recherche d'un état global consistant. Cependant, leur approche n'est pas tolérante aux fautes du fait de l'enregistrement des messages dans les canaux de communication. Typiquement dans le checkpointing, seuls les contextes des processus sont sauvegardés.

Koo et Toueg [64] ont proposé un protocole qui améliore celui proposé par Chandy et Lamport. Le protocole vise a éliminer les messages orphelins en établissant des checkpoints localement de façon convenable. L'état des canaux n'est plus sauvegardé, comparativement au modèle de Chandy et Lamport. Dans cette approche, les processus sauvegardent deux types de checkpoints sur support stable : les checkpoints permanents et les tentatives de checkpoints. Un checkpoint permanent ne peut être annulé alors qu'une tentative de checkpoint peut être soit annulée soit transformée en checkpoint permanent.

Le protocole fonctionne en deux phases pour assurer le fait que soit tous les processus ont fait un checkpoint, soit aucun processus n'a fait de checkpoint. Pour réaliser cela, les deux types de checkpoints sont utilisés. S'il s'avère que l'ensemble des tentatives de checkpoint forme un état global consistant, alors ces tentatives de checkpoints sont transformées en checkpoints permanents.

L'algorithme de Koo et Toueg fonctionne comme suit : dans la première phase, le processus initiateur, disons P, fait une tentative de checkpoint et invite tous les autres à en faire autant. A la réception du message de P, un processus Q établit une tentative de checkpoint. Q peut ne pas le faire pour certaines raisons locales. Le processus Q informe l'initiateur de sa décision et arrête toute communication avec quelque processus que se soit jusqu'à ce que la deuxième phase soit terminée. Si le processus initiateur P est informé que tous les processus ont fait une

tentative de checkpoint, alors dans la deuxième phase, il transforme sa tentative de checkpoint en checkpoint permanent et invite les autres processus à faire autant. Ceci assure que soit tous les processus font un checkpoint permanent, soit aucun ne le fait.

Dans ce protocole il y a possibilité de perte de message. En effet il est possible que Q ait envoyé un message à P avant de faire une tentative de checkpoint, et que ce message soit dans les canaux de communication lorsque P sauvegarde un checkpoint permanent. Toutefois la perte de message n'est pas un obstacle en soit pour l'établissement d'un état consistant, puisque les messages perdus peuvent être rejoués, étant donné qu'ils n'auront pas été enregistrés dans le checkpoint de leur émetteur comme transmis.

L'approche de Koo et Toueg assure que l'état obtenu après que les processus aient tous fait un checkpoint permanent est un état consistant du système, en ce sens qu'il n'y a pas de message orphelin. Une approche similaire au protocole de Koo et Toueg se trouve dans [4].

L'inconvénient majeur de l'approche coordonnée est le surcoût dû à la synchronisation entre les processus pour rechercher un état global consistant. Cependant cette approche est largement utilisée de nos jours, avec le développement de plus en plus poussé des réseaux haute performance qui tendent à réduire l'impact des communications lors de la synchronisation pour la sauvegarde d'une application parallèle.

Les mécanismes de sauvegarde/reprise d'applications sont implémentés à plusieurs niveaux selon les besoins de leur utilisation. Ces différents niveaux sont présentés dans la section suivante.

3.3 Différents niveaux d'implémentation du checkpointing

Il existe trois types d'implémentations du checkpointing : implémentation de niveau application, implémentation de niveau utilisateur et implémentation de niveau système. Ces types diffèrent au niveau de la transparence, de l'efficacité et des mécanismes déployés pour assurer les mécanismes de sauvegarde et de reprise. La transparence dans ce contexte fait allusion au fait que le checkpointing soit effectué sans modification du code source de l'application.

3.3.1 Checkpointing de niveau application

Le checkpointing de niveau application est réalisé par le programmeur même. Le programmeur développe un ensemble de procédures qui assurent les opérations de sauvegarde et de reprise. Cette approche est la plus efficace, car le programmeur a une parfaite connaissance du code de l'application. Il peut ainsi sauvegarder uniquement les données importantes. Ceci a pour intérêt de réduire la taille des points de reprise et les latences. Une heuristique pour un algorithme de placement efficace de points de prise de checkpoint dans les applications a été proposée dans [65]. Cependant cette approche est la moins transparente et requiert plus d'efforts de la part du programmeur.

3.3.2 Checkpointing de niveau utilisateur

Le checkpointing de niveau utilisateur est réalisé au travers de bibliothèques de checkpointing. Cette approche n'implique aucune modification des codes sources du noyau UNIX. Cependant, dans certains systèmes implémentés dans cette approche, l'utilisateur peut être amené à modifier ou recompiler son code source avec la bibliothèque afin de le rendre *checkpointable*. Le checkpoint est établi par envoi de signaux à l'application. Le principal inconvénient dans cette approche est très souvent l'interaction avec le code source de l'application, ce qui la rend moins transparente. Des approches récentes permettent de faire le checkpointing au moyen de bibliothèques mais de façon plus transparente, c'est-à-dire sans qu'il y ait besoin de recompiler les applications.

3.3.3 Checkpointing de niveau système

Le checkpointing de niveau système encore connu sous le nom de checkpointing de niveau noyau est l'approche la plus transparente. Dans cette approche, les routines chargées d'assurer la sauvegarde et la reprise sont développées sous forme de modules qui sont chargés dans le noyau du système d'exploitation. Cette approche n'induit aucune modification du code source de l'application. Cependant, cette approche est la moins efficace. Les seules optimisations possibles sont faites au niveau du système de checkpointing. Le checkpointing de niveau système est moins contraignant et très adapté pour les gestionnaires de ressources, qui dans la plupart du temps ne gèrent que les exécutables des applications. En général certains privilèges sont requis juste pour l'installation de tels systèmes.

Pour le checkpointing d'applications séquentielles, nous utiliserons une implémentation du checkpointing de niveau système, car cela cadre bien avec les objectifs des ordonnanceurs de tâches qui ne manipulent que les exécutables des applications. Pour les applications parallèles nous utiliserons une implémentation du checkpointing coordonné.

Dans la section qui suit, nous présentons quelques systèmes de checkpointing, chacun appartenant à l'un des niveaux précédemment mentionnés.

3.4 Quelques systèmes de checkpointing

De nombreuses implémentations des mécanismes de sauvegarde/reprise basées sur les approches présentées dans les sections 3.2 et 3.3 ont été proposées. Dans la plupart des cas, ces systèmes sont présentés sous forme de logiciel libre avec des codes sources disponibles et sont exploités à des fins de recherche. Quelques versions commerciales existent sur le commerce.

Dans cette section, nous présentons les caractéristiques de quelques systèmes de sauvegarde/reprise d'applications séquentielles et d'applications parallèles.

3.4.1 Systèmes de checkpointing séquentiels

Ces systèmes sont conçus pour effectuer la sauvegarde et la reprise d'applications mono-processus.

Condor

Le projet Condor développé à l'Université de Wisconsin offre une bibliothèque de sauvegarde/reprise [5] qui peut être utilisée séparément de l'environnement Condor. Cette bibliothèque, de niveau utilisateur, porte également le nom de Condor, tout comme le projet. Condor est conçu pour la sauvegarde/reprise des applications *monothread* . Pour faire un checkpointing, l'utilisateur doit compiler son code source pour le lier avec la bibliothèque de checkpointing de Condor. Cette bibliothèque installe tout d'abord un manipulateur de signaux pour la sauvegarde de l'image d'un processus de façon asynchrone, ensuite elle fournit au noyau un ensemble d'appels systèmes pour la restauration ou la migration de processus. La sauvegarde est engagée lorsque le processus reçoit le signal SIGSTP. A la réception de ce signal, un processus fils est crée pour enregistrer le contexte d'exécution du processus pendant que ce dernier continue son exécution. Condor peut être configuré pour effectuer des checkpoints périodiques. Une des restrictions de Condor se situe au niveau de la portabilité. En effet Condor est limité à fonctionner sur certaines architectures. Condor n'est, par exemple, pas compatible avec les machines Opteron.

Libckpt

Libckpt [66] implémente un checkpointing de niveau utilisateur moins transparent que Condor. Il est restreint à la sauvegarde des applications *monothread*. En fait le code source de l'application nécessite quelques modifications avant que le checkpointing ne puisse s'opérer. Par exemple le *main()* dans un programme en C doit être changé en *ckpt_target()*. Ceci permet à Libckpt de prendre contrôle de l'application au début de son exécution.

Cryopid

Cryopid [67] est un système de checkpointing de niveau utilisateur qui permet de sauvegarder le contexte d'exécution des processus *monothread* GNU/Linux. Cryopid offre plus de transparence encore que Condor, en ce sens que l'utilisateur n'a pas besoin de recompiler ou de relinker son application. Lors de la sauvegarde d'une application, Cryopid génère un checkpoint qui est un binaire contenant tout le nécessaire pour la reprise de l'application. A la reprise, le checkpoint est redémarré comme tout exécutable. Un des problèmes des anciennes versions de Cryopid (2005) était la taille considérable du checkpoint généré. Les versions actuelles intègrent des mécanismes de compression qui tendent à réduire cette taille.
Nous avons utilisé Cryopid au moment où nous commencions cette thèse pour deux raisons : sa disponibilité et surtout parce qu'il présentait cette particularité de pouvoir générer le point de

reprise comme un exécutable. Cela offrait des éléments de comparaison avec d'autres systèmes de checkpointing.

MTCP

MTCP (**M**ulti**T**hreaded **C**heck**P**ointing) [68] est un système de checkpointing de niveau utilisateur transparent qui permet la sauvegarde et la reprise d'applications séquentielles *multithreads*. MTCP offre un niveau de transparence plus élevé que Condor. En effet les versions récentes de MTCP utilisent l'architecture ELF (Executable and Linking Format) [69] pour injecter les routines de sauvegarde/reprise dans l'exécutable de l'utilisateur. Cette transparence est semblable à celle fournie par les systèmes de checkpointing de niveau noyau.

BLCR

BLCR (Berkeley Lab Checkpoint/Restart) [70, 71] est un système de checkpointing de niveau noyau ou système développé dans les laboratoires de l'université de Berkeley. BLCR offre une interface niveau utilisateur qui permet de faire abstraction de la complexité du noyau. Des privilèges d'administrateurs sont nécessaires juste pour son installation. Outre son aptitude à effectuer la sauvegarde et la reprise d'applications séquentielles multithreads, un autre atout de BLCR est sa capacité à tourner sur un large pannel d'architecture de machines. Ceci a fait de BLCR l'un des systèmes de checkpointing les plus largement utilisés. Certains environnements d'exécution d'applications parallèles MPI comme LAM/MPI, MPICH-V, MVAPICH, Open MPI utilisent BLCR comme support de sauvegarde au niveau de chaque processus de l'application parallèle. BLCR peut être également utilisé dans bon nombre de gestionnaires de ressources pour assurer l'équilibrage des charges à travers la migration de processus.
Nous avons utilisé et adopté BLCR comme système de checkpointing du fait de sa portabilité dans une diversité d'architectures de machines et du fait de ses performances comme on le verra à la section 3.5.

3.4.2 Systèmes de checkpointing parallèles

Ces systèmes sont conçus pour effectuer la sauvegarde et la reprise d'applications qui s'exécutent au travers de plusieurs processus. Généralement, Les systèmes de checkpointing parallèles s'appuient sur les systèmes de checkpointing séquentiels pour effectuer la sauvegarde globale de l'application. Dans cette section nous présentons quelques uns de ces systèmes.

CoCheck

Le système CoCheck [72] offre des mécanismes de sauvegarde/reprise d'applications parallèles dans l'environnement Condor. CoCheck utilise la bibliothèque Condor pour sauvegarder l'état d'un processus. Pour la sauvegarde d'un état global consistant de l'application parallèle, CoCheck utilise les techniques de sauvegarde coordonnée basées sur l'approche de l'algorithme

de Chandy et Lamport [2]. Une des contraintes de CoCheck est que en cas de reprise d'une application ou d'un processus à partir de son checkpoint sur une nouvelle ressource, cette dernière doit avoir la même configuration que celle à partir de laquelle le checkpoint a été effectué.

Charm++

Charm++ [73] est un environnement de programmation parallèle par objet basé sur le langage C++. La description du parallélisme dans Charm++ est basée sur la création d'objets concurrents appelés chares. Un objet chare est un objet C++ standard dont les méthodes peuvent être appelées à distance par un autre chare disposant d'une référence sur cet objet.

Charm++ utilise deux approches pour assurer la sauvegarde coordonnée d'applications. La première approche consiste à sauvegarder les checkpoints sur supports stables. Les checkpoints sont les chares qui sont empaquetés sous forme de fichiers et sauvegardés sur disque. L'état d'un processus sauvegardé est l'ensemble des chares qu'il contient. Cette approche est utilisée pour assurer la migration de processus dans Charm++ en vue d'équilibrer les charges dans l'environnement. Cela permet également aux utilisateurs de sauvegarder leurs applications pour une reprise ultérieure. L'autre approche consiste à sauvegarder les points de reprise de chaque processus en mémoire, en vue d'assurer la tolérance aux fautes. Lorsqu'une panne survient, la reprise globale de tous les processus est imposée par la sauvegarde coordonnée. Afin de réduire le surcoût d'une reprise, les points de reprise de chaque processus sont stockés en mémoire locale puis répliqués sur la mémoire d'une autre unité de calcul qui est considérée comme un serveur de stockage pour le premier processus. Bien que efficace, cette approche induit un surcoût d'utilisation de la mémoire.

Une extension de Charm++, AMPI [74] (Adaptative MPI) a été conçue pour assurer la répartition automatique des charges, apporter plus de recouvrement de communication et une flexibilité d'exécution d'applications sur un nombre dynamique de processeurs dans l'environnement.

MPICH-V

MPICH-V est un projet de recherche développé à l'université de Paris XI-Orsay qui implémente plusieurs protocoles de tolérance aux fautes du standard MPI [75]. MPICH-V1 est basé sur la sauvegarde non coordonnée avec journalisation des messages pour des programmes parallèles écrits avec MPI. Afin de pouvoir redémarrer uniquement les processus défaillants, MPICH-V1 utilise le concept de canal mémoire ; tous les messages échangés passent par un canal mémoire supposé fiable. Cette technique implique qu'il n'est pas possible d'avoir de communications directes entre les processus. Pour éviter l'utilisation du canal mémoire, MPICH-V/Causal implémente une journalisation causale des événements non déterministes. MPICH-V/CL supporte un protocole de sauvegarde coordonnée. Afin d'améliorer la performance d'une reprise globale, chaque processus stocke ses points de reprise en local et sur mémoire stable.

Les versions actuelles de MPICH-V utilisent les bibliothèques de points de reprise fournies

par BLCR, Condor et ckpt [1] pour la sauvegarde individuelle des processus. De ces trois systèmes, MPICH-V accorde plus de privilège à BLCR du fait qu'il puisse être opérationnel sur un large pannel de types de machines.

LAM/MPI

LAM/MPI [76] est une implémentation du standard MPI qui utilise l'approche coordonnée pour la sauvegarde et la reprise d'applications parallèles. LAM/MPI comprend une couche qui offre des mécanismes de sauvegarde/reprise. Cette couche est totalement indépendante de celle utilisée pour la programmation et l'exécution d'applications MPI. LA sauvegarde d'une application avec LAM/MPI se fait en deux phases. Dans la première phase, les processus synchronisent leurs états pour trouver un état global consistant. A l'issue de cette phase les canaux de communication entre les processus sont vides. Cette approche qui consiste à vider les canaux de communication induit un surcoût considérable sur le temps de sauvegarde des applications. Dans la seconde phase, LAM/MPI utilise BLCR pour sauvegarder chaque processus individuellement. A la reprise de l'application, tous les processus reprennent leur exécution à partir de leurs points de reprise sauvegardés, avec les canaux de communication restaurés à l'état vide. LAM/MPI n'est plus maintenu depuis 2008 car toutes les compétences sont orientées vers le projet fédérateur Open MPI.
LAM/MPI a été le système de checkpointing retenu pour nos tests dans le cas des applications parallèles, du fait de sa très forte compatibilité avec BLCR.

Open MPI

Open MPI [77, 78] est un projet fédérateur de quatre implémentations MPI : FT-MPI de l'université du Tennessee, LA-MPI du laboratoire national de Los Alamos, LAM/MPI de l'université d'Indiana et de PACX-MPI de l'université de Stuttgart. La motivation principale de Open MPI est d'agréger les meilleures idées et expertises de ces différents projets pour créer une implémentation du standard MPI qui puisse être opérationnel sur un large pannel d'architectures de machines et d'applications. Open MPI est ainsi une implémentation du standard MPI extensible et modulaire, intégrant des mécanismes de tolérance aux fautes par des points de reprise coordonnés. Tout comme un bon nombre de système libre, Open MPI reste ouvert à toute contribution.

DejaVu

DejaVu [79] est un système de checkpointing de niveau utilisateur qui effectue la sauvegarde/reprise coordonnée. Ce système offre un niveau de transparence qui est également semblable à celle fournie par les systèmes de checkpointing de niveau noyau. Contrairement à LAM/MPI, DejaVu sauvegarde l'état des sokets et des canaux de communication lors d'une sauvegarde, ce

1. http ://pages.cs.wisc.edu/~zandy/ckpt/

qui a pour but de minimiser l'impact des communications sur la sauvegarde globale de l'application. DejaVu utilise des optimisations pour minimiser le volume de données à sauvegarder dans les images.

DMTCP

DMTCP [80] est un système de checkpointing de niveau utilisateur transparent qui effectue la sauvegarde/reprise coordonnée d'applications parallèles. Contrairement aux systèmes précédents qui sont basés pour la plupart sur la bliothèque MPI, DMTCP ne dépend pas d'une bibliothèque de passage de message particulière. Ceci donne à DMTCP un large champ d'utilisation. Pour la sauvegarde d'applications, deux couches de DMTCP collaborent. La première s'occupe de la sauvegarde des informations relatives aux interactions entre les processus puis informe la seconde qui fait appel à MTCP pour la sauvegarde individuelle de chaque processus.

3.4.3 Bilan sur les systèmes de checkpointing

La sauvegarde/reprise d'application est une technique de plus en plus maîtrisée, mais il convient de signaler qu'aucun système de checkpointing ne peut encore effectuer la sauvegarde/reprise de tout type d'application. Ceci vient premièrement de ce que ces environnements sont des projets de recherche, et deuxièmement de ce que en général le développement du logiciel ne suit pas celui du matériel qui va plus vite.

Comme nous pouvons le constater, la sauvegarde/reprise d'applications est un sujet largement étudié depuis plusieurs décennies et de nombreuses implémentations ont été proposées. Dans nos travaux, nous ne nous attèlerons donc plus à proposer de nouvelles techniques de sauvegarde/reprise. Nous utiliserons les implémentations existantes pour effectuer la sauvegarde et la reprise d'applications dans le cadre de nos études. Plus précisément, nous utiliserons les systèmes BLCR pour la sauvegarde d'applications séquentielles et LAM/MPI pour la sauvegarde d'applications parallèles. Le choix de ces systèmes de checkpointing est conséquent de la présentation faite sur ces systèmes.

Comme nous l'avons présenté, les systèmes de checkpointing offrent des performances variées en fonction de leur conception et de leur implémentation. Ces performances peuvent être influencées par certains paramètres lors du checkpointing. Dans la section qui suit, quelques uns de ces paramètres sont présentés.

3.5 Quelques paramètres importants du checkpointing

Dans cette section nous présentons quelques tests effectués avec quelques systèmes de checkpointing. Nous ne décrirons pas ici entièrement l'environnement expérimental, ceci sera fait dans le chapitre suivant. Le but ici est juste de faire ressortir quelques paramètres importants lors d'une sauvegarde d'application et également montrer l'influence que le système de checkpoin-

ting peut avoir lors d'une sauvegarde. Pour ce dernier cas nous comparerons les performances de deux systèmes de checkpointing à savoir BLCR et Cryopid.

3.5.1 Importance du volume de données à sauvegarder

Le temps de sauvegarde d'une application séquentielle ou parallèle est fortement dépendant du volume de données à sauvegarder, comme nous allons le voir dans le cas de la sauvegarde d'applications séquentielles et parallèles.

Cas d'une application séquentielle

L'application utilisée pour cette expérience est un code de calcul multigrilles développé au laboratoire ID-IMAG [2] pour la résolution des équations aux dérivées partielles. C'est une application dont la taille en mémoire croît considérablement lorsque l'ont fait accroître le nombre de points sur les grilles.

La figure 3.1 montre que le temps de sauvegarde croît lorsque le volume de données augmente, comme on pouvait s'y attendre. Le système de checkpointing utilisé ici est BLCR.

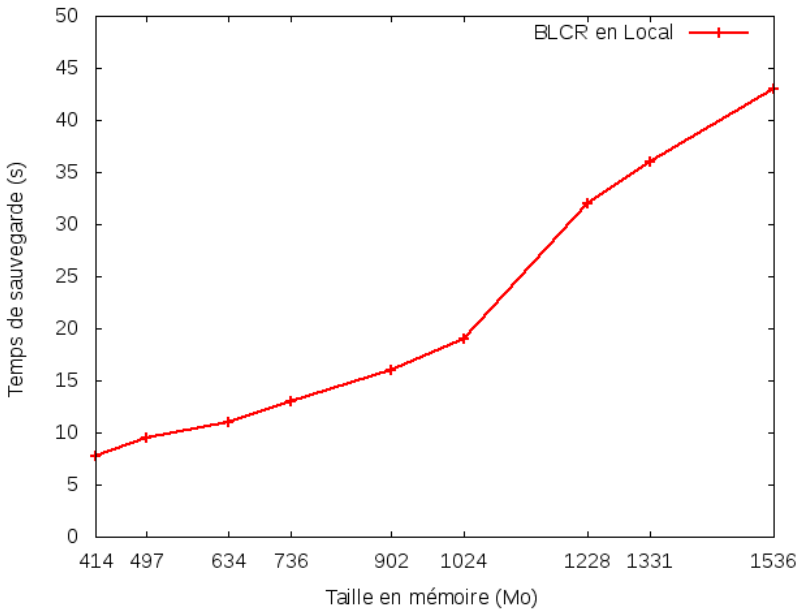

FIGURE 3.1 – Temps de sauvegarde en fonction du volume de données à sauvegarder : cas d'une application séquentielle

2. devenu LIG (Laboratoire d'Informatique de Grenoble)

Cas d'une application parallèle

Pour ce test nous avons utilisé le système de checkpointing parallèle LAM/MPI (version 7.1.4). L'application sauvegardée est le code de calcul du nombre *pi*. Ce code, fourni avec pratiquement toutes les implémentations du standard MPI, est modifié de sorte que toutes les pages mémoires soient sauvegardées. Le test est effectué dans une grappe dont le nombre de noeuds est choisi en fonction du nombre de processus de l'application parallèle.

La taille en mémoire est la même pour chaque processus de l'application qui s'exécute avec un processus par noeud. La figure 3.2 montre que le temps de sauvegarde croît proportionnellement avec le nombre de noeuds et la taille en mémoire par noeud, donc avec le volume de données à sauvegarder.

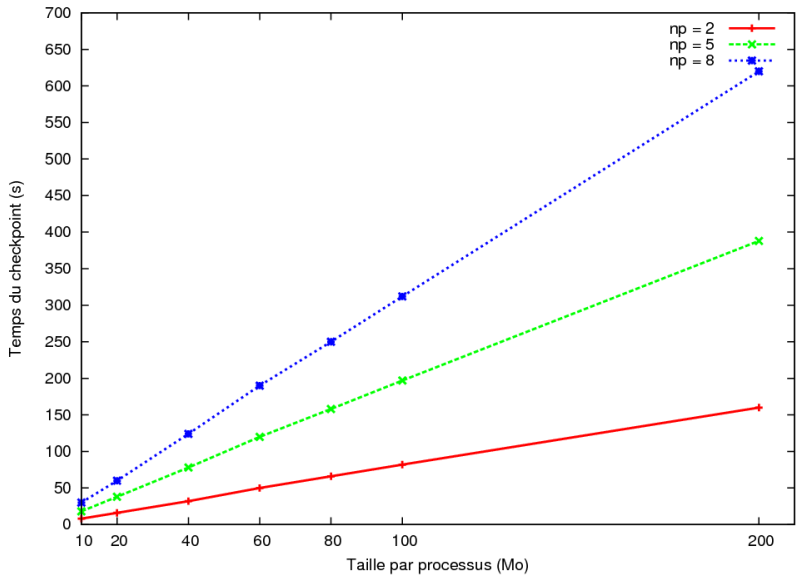

FIGURE 3.2 – Temps de sauvegarde en fonction du volume de données à sauvegarder : cas d'une application parallèle

3.5.2 Influence de la proximité du support de sauvegarde

Nous utilisons BLCR pour comparer le temps de sauvegarde en local et à distance. Le code séquentiel utilisé est le même que celui de la section 3.5.1. La figure 3.3 montre que la sauvegarde à distance a un coût de loin plus élevé en temps que la sauvegarde en local. Cela se comprend car les données transitent d'abord vers le réseau avant d'arriver à leur destination finale.

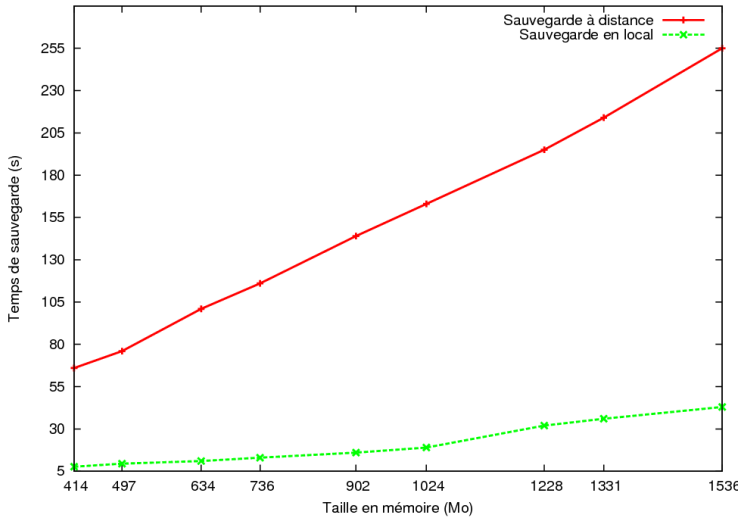

FIGURE 3.3 – Temps de sauvegarde en fonction de la proximité du support de sauvegarde

3.5.3 Influence du système de checkpointing

L'expérience présentée ici montre l'impact que le système de checkpointing peut avoir sur l'exécution de l'application lors de la sauvegarde. Nous comparons la sauvegarde de l'application du calcul des fractales de Mandelbrot réalisée par les systèmes de checkpointing BLCR (version 0.4.2) et Cryopid (Version 0.5.9). Cette application est séquentielle et la taille qu'elle occupe en mémoire varie en fonction du nombre de points fournis pour le calcul des fractales. La sauvegarde est effectuée sur le disque local.

La figure 3.4 montre que le temps de sauvegarde réalisé par BLCR est de loin inférieur à celui obtenu avec Cryopid. Ceci permet de déduire que l'exécution de l'application peut être affecté par le système de checkpointing utilisé. Ainsi en fonction du système de checkpointing utilisé, le surcoût dû au checkpointing peut être élevé ou non.

La figure 3.5 montre une grande disparité entre les tailles des points de reprises générés avec BLCR et Cryopid. Ceci montre également que le type de système de checkpointing utilisé peut affecter plus ou moins l'espace de stockage.

3.5.4 Synthèse

Des études menées, nous constatons que, outre le type de système de checkpointing utilisé, les performances du checkpointing sont influencées par le volume de données à sauvegarder et le chemin de sauvegarde. Ceci permet de déterminer les paramètres importants à prendre en compte lors d'une étude de modélisation d'un environnement de sauvegarde.

Dans les sections précédentes, nous avons vu comment les applications sont sauvegardées et restaurées lors d'une reprise. Nous avons également présenté quelques systèmes de sauvegarde/-

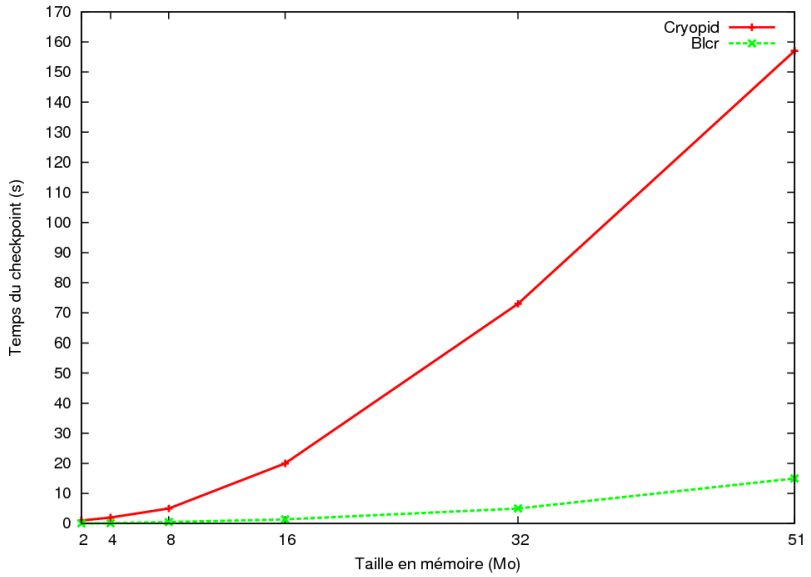

FIGURE 3.4 – Temps de sauvegarde réalisé avec BLCR et Cryopid sur le code Mandelbrot

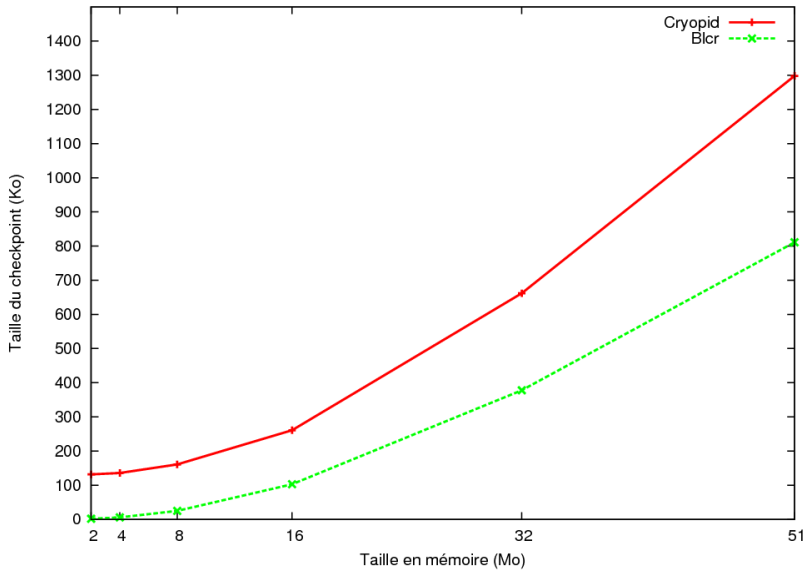

FIGURE 3.5 – Taille du point de reprise généré par BLCR et Cryopid sur le code Mandelbrot

reprise qui implémentent les mécanismes du checkpointing avec des approches appropriées à des contextes. Cependant, la sauvegarde d'une application peut induire des pénalités sur l'exécution de l'application, et une occupation excessive de l'espace de stockage des points de reprise. Nous discutons de cela dans la section qui suit.

Nous assimilons la reprise d'une application à son initialisation. De ce fait la reprise d'application ne fera pas l'objet d'une étude dans nos travaux. De même nous supposons l'espace

de stockage suffisamment grand pour la sauvegarde des points de reprise. Ainsi nous excluons également l'espace de stockage du champ de nos travaux.

3.6 Performances du checkpointing

La finalité du checkpointing est de pouvoir sauvegarder correctement le contexte des processus en vue de leur reprise. Il convient de souligner que le checkpointing peut avoir un impact sur l'environnement où il est invoqué, d'où la nécessité d'accorder une attention particulière sur la performance du checkpointing. Plusieurs métriques permettent de mesurer la performance du checkpointing : le *surcoût du checkpointing*, le *temps de sauvegarde*, le *temps de reprise* et l'*espace de stockage occupé*.

Comme nous l'avons dit précédemment, nous ne discutons pas dans le cadre de cette thèse du temps de reprise et de l'espace de stockage occupé.

Le surcoût du checkpointing (Figure 3.6) est le temps ajouté au temps d'exécution de l'application. C'est le temps nécessaire pour initier une opération de sauvegarde/reprise. Durant cette phase l'exécution de l'application est suspendue et donc n'effectue pas de calcul, ce qui peut être considéré comme une perte de temps pour l'application. Le surcoût du checkpointing est donc la métrique la plus importante.

Le temps de sauvegarde Ts (Figure 3.6) inclut généralement le surcoût Tc et le temps d'écriture sur support stable ($Ts - Tc$) pendant que l'application poursuit son exécution. On parle alors dans ce cas de *checkpointing non bloquant*. Lorsque le surcoût est égal au temps de sauvegarde, on parle de *checkpointing bloquant*. Dans ce cas la reprise de l'application ne s'effectue qu'après la sauvegarde complète du point de reprise. Une étude menée dans [81] montre que dans les grappes et les grilles de calcul, le surcoût induit par l'approche du checkpointing coordonné bloquant est très élevé comparativement à celui induit par l'approche du checkpointing coordonné non bloquant.

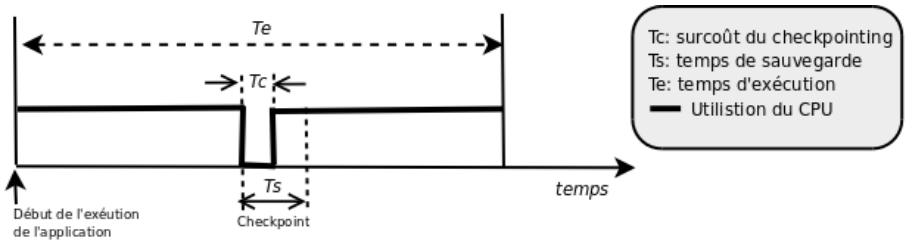

FIGURE 3.6 – Séquence de prise de point de reprise au cours de l'exécution d'une application

La performance du checkpointing peut être étudiée à deux niveaux : au niveau de l'implémentation des systèmes de sauvegarde/reprise et au niveau de l'utilisation des systèmes de sauvegarde/reprise.

3.6.1 Performance du checkpointing au niveau de l'implémentation

Plusieurs méthodes sont utilisées pour minimiser le surcoût du checkpointing et/ou le volume de données à sauvegarder. Nous présentons quelques unes les plus utilisées.

Une approche consiste à utiliser les primitives de mémoire virtuelle pour ne sauvegarder que les pages mémoires qui ont été modifiées entre l'instant où l'application a été suspendue pour initier un checkpoint et l'instant où les pages sont effectivement écrites sur support stable. Cette approche qui minimise le surcoût du checkpointing (car l'application reprend plus vite son exécution pendant que l'écriture des points de reprise se poursuit) est connue sous le nom de *copy-on-write buffering*. Elle est utilisée par les systèmes comme BLCR et LAM/MPI.

Une autre approche consiste à ne sauvegarder que les pages mémoires qui ont changé entre deux checkpoints. Cette approche est indiquée lorsque les points de reprise doivent être pris très régulièrement. Elle vise à minimiser le surcoût du checkpointing et le volume de données à sauvegarder. De nombeux algorithmes ont été proposés dans cette approche connue sous le nom de *checkpointing incrémental* [82, 83, 84]. Cette approche est utilisée par les système comme MTCP, Deja-vu, DMTCP.

Une autre méthode consiste à utiliser des algorithmes de compression de données pour minimiser le volume de données à sauvegarder et ainsi réduire le temps des sauvegardes. Cette technique est utilisée par Cryopid.

3.6.2 Performance du checkpointing au niveau de l'utilisation

Notre intérêt pour l'étude de la performance du checkpointing se situe au niveau de l'utilisation des systèmes de sauvegarde/reprise. De nombreux travaux de recherche [85, 86, 87, 88, 89, 90] ont contribué à améliorer les performances du checkpointing, en mettant l'accent sur la totalité ou une partie des caractéristiques des systèmes distribués comme la disponibilité, la fiabilité et la maintenabilité.

J. S. Plank et M. G. Thomason [86, 87] ont proposé un modèle dans lequel la disponibilité moyenne A du système est une mesure de performance du *checkpointing*. Ce paramètre donné par la formule mathématique $A = \left(\left(I - Ce^{-\lambda I} \right) \frac{e^{-\lambda I}}{1 - e^{-\lambda I}} \right) \lambda e^{-\lambda(R+L)}$ est maximisé par la détermination d'un intervalle I optimal entre deux sauvegardes. I est la fréquence de *checkpointing* qui minimise les pertes de temps pour une application qui s'exécute en présence de pannes, λ est le nombre moyen d'apparition des pannes, C est le surcoût du checkpointing, L le temps de sauvegarde et R le temps de reprise. L'idée qui se dégage de cette formule c'est la minimisation du surcoût du checkpointing, car en effet plus le système est disponible, plus les applications passent le temps à effectuer effectivement des calculs. Leur modèle ne tient pas compte explicitement de l'espace mémoire occupé par l'application, lequel influence considérablement le temps de sauvegarde.

Dans [89, 90] des modèles théoriques pour la détermination d'un intervalle optimal en vu d'un placement efficace des points de reprise sont proposés dans les environnements où les

applications s'exécutent avec des arrivées probabilistes de pannes dans le système. L'objectif ici est de minimiser le surcoût du checkpointing induit par des sauvegardes régulières pour assurer la tolérance aux fautes. Ces modèles sont basés sur un grand nombre d'hypothèses qui ne prennent pas en compte les applications et les environnements matériels.

E. Imamagic et al. [85] ont proposé une approche pour optimiser les performances du checkpointing en minimisant les temps de sauvegarde des points de reprise et la charge du réseau. Ils considèrent deux intervalles de temps. La plus courte période correspond au temps d'exécution entre deux points de reprise qui doivent être sauvegardés localement. La plus longue correspond au temps d'exécution entre deux points de reprise qui doivent être sauvegardés sur un support distant. L'approche de sauvegarde locale permet ainsi de minimiser la charge du réseau et d'accroître la vitesse des sauvegardes. Cette approche est certes efficace, mais elle n'est indiquée que pour des grappes dédiées. De plus leur modèle ne prend pas en compte les conflits qui peuvent se présenter pour les accès concurrents au réseau.

La plupart des travaux visant à améliorer les performances du checkpointing considèrent des sauvegardes à intervalles réguliers, sans trop se préoccuper de l'environnement matériel. Dans le cadre de nos travaux, nous étudions les performances du checkpointing en tenant compte des contraintes réseau et disque pour minimiser les pertes de temps de sauvegarde et gagner en temps de calcul. Dans notre étude nous prenons en compte le volume de données à sauvegarder, donné par l'espace mémoire occupé par les applications. Les détails de cette étude sont fournis dans le chapitre suivant.

Dans le souci d'améliorer les performances du checkpointing, et au vu des paramètres présentés à la section 3.5, plusieurs stratégies de sauvegarde ont été adoptées et implémentées dans les environnements dynamiques. Dans la section suivante, nous examinons quelques unes de ces stratégies.

3.7 Stratégies de sauvegarde dans les environnements dynamiques

L'étude des performances du checkpointing montre que la sauvegarde du contexte d'exécution d'applications peut créer un surcoût pénalisant sur le temps d'exécution des applications. Dans le but de permettre aux applications de s'exécuter sans trop de perturbation dans les environnements dynamiques, des stratégies de sauvegarde ont été proposées.

3.7.1 Différentes stratégies

Dans [91] plusieurs stratégies pour la sauvegarde distribuée des points de reprise ont été présentées pour les environnements de calcul où les ressources sont non dédiées. Dans ces environnements les machines peuvent tomber en panne et passent très fréquemment d'état libre à état occupé, compromettant l'évolution des applications qu'elles exécutent. Pour permettre à ces systèmes d'être tolérant aux fautes, [91] propose de sauvegarder les points de reprise sur

des machines de l'environnement avec plusieurs approches.

Une stratégie consiste à répliquer les points de reprise. Le système sauvegarde une copie du point de reprise localement et une autre sur un support distant. L'avantage est que, si un replicat devient inaccessible, le système peut utiliser l'autre tant que le nœud qui le contient est disponible.

Une alternative à la réplication consiste à appliquer les algorithmes de parité [92] pour diviser le point de reprise de taille n en m fragments de taille $\frac{n}{m}$ sauvegardés sur plusieurs supports distants avec un fragment additionnel qui contient les informations de parité. Cette approche permet d'éviter l'encombrement des supports distants car n'y est sauvegardée qu'une fraction du point de reprise.

Une stratégie plus efficace que la précédente consiste à appliquer l'algorithme de Michael Rabin [93] pour encoder un point de reprise de taille n en $m+k$ fragments de taille $\frac{n}{m}$ sauvegardés sur plusieurs supports distants. Cette approche permet de tolérer k fautes dans le système. L'inconvénient de cette approche est le surcoût du à la complexité de l'algorithme d'encodage.

Dans les trois précédentes approches, les points de reprise sont d'abord sauvegardés localement sur la machine hôte avant d'être transférés sur un support distant. Ces approches ne peuvent pas être utilisées dans notre contexte car nous avons opté travailler dans des environnements dynamiques où l'indisponibilité d'un poste de travail implique que ses ressources (en particulier le disque) ne sont plus utilisables. De plus l'approche de sauvegarde locale suivie d'un transfert vers un support distant induit un surcoût du checkpointing.

Une autre stratégie présentée dans [91] consiste à sauvegarder les points de reprise vers un support centralisé. Cette stratégie peut s'appliquer dans le contexte de grappes virtuelles que nous avons adopté. Mais dans [91], les points de reprises sont sauvegardés vers un support centralisé sans se préoccuper des goulots d'étranglement qui peuvent se former au niveau du support centralisé, ce qui pourrait fortement induire un surcoût du checkpointing.

3.7.2 Prédiction de disponibilité des ressources

Dans le but de rendre les environnements dynamiques de calcul tolérants aux fautes, [94] propose des sauvegardes régulières des points de reprise sur des postes de travail qui auraient fourni leurs disques comme espace de stockage, les points de reprise étant pris sur les postes de travail qui n'auraient fourni que les temps de cycle CPU comme ressource. Ces derniers ne sont alors utilisés que comme noeuds de calcul.

Pour la sauvegarde des points de reprise, l'utilisation intensive du réseau pourrait avoir une incidence négative sur les performances du système global. Pour éviter cela, les auteurs de [94] proposent de sélectionner un ensemble de support de sauvegarde parmi les noeuds s'étant proposés comme serveurs de stockage en tenant compte à la fois de leur disponibilité future et de leur connectivité réseau. Les supports de sauvegarde sélectionnés sont utilisés pour enregistrer de façon redondante des petits fragments des points de reprise générés par l'encodage des points de reprise après application de l'algorithme proposé dans [95].

Dans cette approche, les points de reprise sont d'abord provisoirement sauvegardés localement. Ensuite ils sont fragmentés avant d'être déplacés par petits blocs vers les serveurs de sauvegarde pour un stockage permanent. Cette situation est identique à celle présentée à la section 3.7.1, ce qui rend cette approche inappropriée à notre contexte d'utilisation des ressources libres d'un intranet.

3.7.3 Stratégie de Condor

Condor utilise les mécanismes de checkpointing pour assurer la tolérance aux fautes face à la volatilité des machines et pour effectuer la migration de processus. Les checkpoints sont pris régulièrement à des intervalles définis par l'utilisateur ou par le système et les points de reprise sont sauvegardés sur des serveurs de stockage dédiés [44]. Pour minimiser le surcoût des sauvegardes, Condor adopte une politique d'organisation de la grappe par domaine d'exécution. Les ressources de chaque domaine sont connectées en interne par un réseau rapide et partagent un serveur de stockage des points de reprise. Les machines qui ne sont pas spécifiquement affectés à un domaine d'exécution effectuent leur checkpoint vers un serveur spécifique de stockage des points de reprise.

Tout comme les précédentes approches, Condor ne se préoccupe pas des goulots d'étranglement qui peuvent se former au niveau du support centralisé, qui pourraient induire un surcoût du checkpointing.

3.7.4 Synthèse

Les stratégies de sauvegarde présentées dans cette section mettent beaucoup plus l'accent sur la tolérance aux fautes dans les environnements dynamiques. Aucun de ces travaux n'étudie les problèmes que peut poser l'écriture simultanée de plusieurs points de reprise vers un même support de sauvegarde. Certes cela pourrait ne pas constituer une priorité pour eux, mais ce problème peut être source de perte de performance du système global. La section qui suit apporte plus de détails à ce conflit.

3.8 Position du problème

Dans le cadre de cette thèse, nous avons opté utiliser les ressources libres d'un intranet comme ressources de calcul intégrées dans un environnement fortement dynamique tel qu'une grappe virtuelle. Pour éviter toute panne qui pourrait survenir suite à l'utilisation du disque d'une ressource de calcul et parce que la ressource pourrait ne plus être disponible à la prochaine sollicitation, nous avons adopté de n'utiliser que la RAM et le CPU des noeuds pour faire du calcul scientifique.

Les postes acquis doivent être restitués au terme de leur période d'inactivité, par exemple à 8 heures au retour des principaux utilisateurs. Il est donc nécessaire de sauvegarder les calculs

inachevés. C'est à ce niveau que se situe notre intérêt pour le checkpointing. Compte tenu de la volatilité des ressources, nous avons opté d'effectuer la sauvegarde du contexte depuis la mémoire sur le noeud de calcul vers le disque du serveur. La figure 3.7 illustre ces propos.

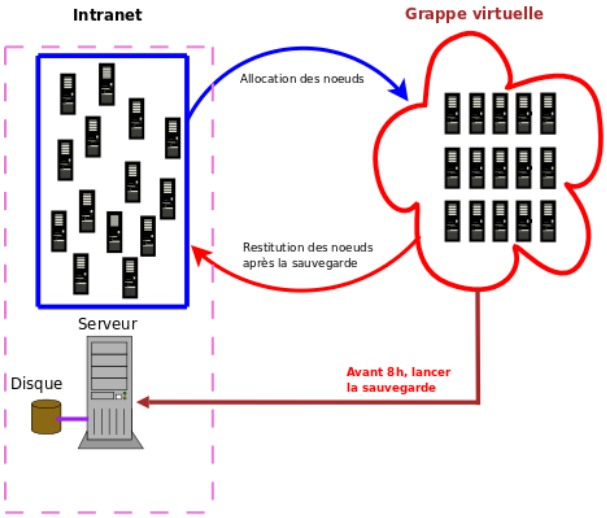

FIGURE 3.7 – Sauvegarde d'applications avant restitution des ressources

Cependant la sauvegarde en parallèle à distance de plusieurs applications peut faire face à plusieurs contraintes. On pourrait certes sauvegarder les applications l'une après l'autre, mais cela entraînerait une mauvaise exploitation des ressources. En effet considérons le cas où 100 applications identiques doivent être sauvegardées séquentiellement. Si une sauvegarde prend 90 secondes, il faudrait plus de 2 heures pour sauvegarder toutes les applications. De ce fait, les premières applications à sauvegarder perdraient près de deux heures dans le temps de disponibilité des ressources. D'où la nécessité de sauvegarder en parallèle.

Dans le cas de la sauvegarde en parallèle à distance de plusieurs applications séquentielles indépendantes, il peut y avoir des congestions au niveau de l'utilisation du réseau, des bus PCI et du disque, comme le montre la figure 3.8.

Dans le cas de la sauvegarde en parallèle à distance de plusieurs applications parallèles, en plus des contraintes liées à l'utilisation du réseau, des bus PCI et du disque, s'ajoutera le surcoût dû à la synchronisation entre les processus des applications, comme vu à la figure 3.9.

3.9 Conclusion

Dans ce chapitre nous avons étudié le problème de la sauvegarde d'applications dans les environnements dynamiques. Nous avons présenté les différentes techniques utilisées et quelques systèmes implémentant ces techniques. Nous avons vu que les mécanismes de sauvegarde/reprise ont été largement utilisés pour assurer la tolérance aux fautes dans les environnements dynamiques, et que notre intérêt pour le checkpointing était plus porté vers la préemption.

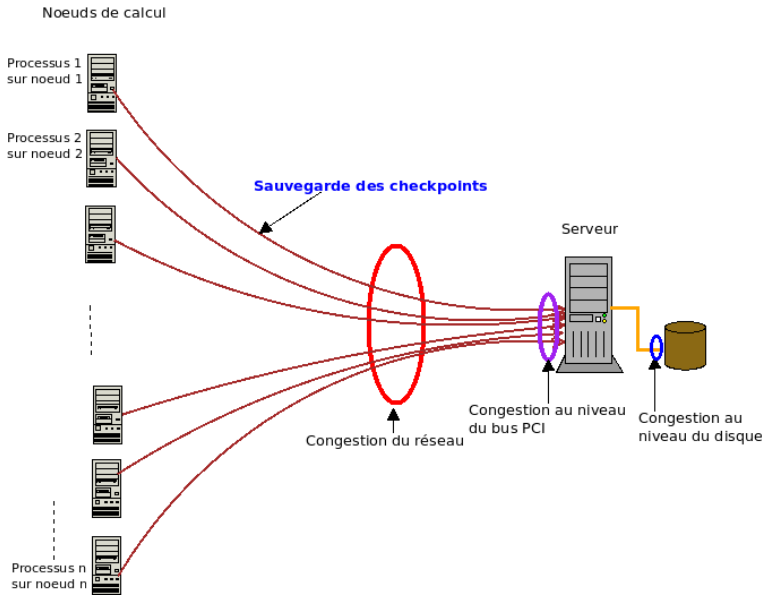

FIGURE 3.8 – Sauvegarde d'applications séquentielles avant restitution des ressources

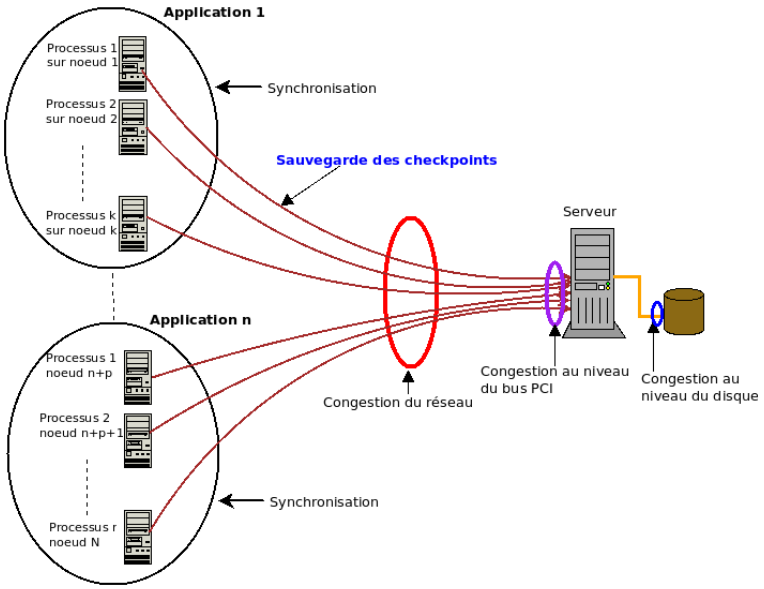

FIGURE 3.9 – Sauvegarde d'applications parallèles avant restitution des ressources

Les stratégies de sauvegarde que nous avons étudiées à la section 3.7 se sont avérées inadaptées à notre contexte d'utilisation des ressources libres d'un intranet comme infrastructure de calcul, car ne prenant pas en compte les problèmes posés par les sauvegardes simultanées.

L'étude menée à la section 3.8 montre l'importance de la sauvegarde en parallèle dans les environnements de calcul de type grappe virtuelle en soulevant les problèmes que cela pose.

Cette stratégie de sauvegarde cadre avec le souci d'utiliser à bon escient les ressources libres d'un intranet, mais il va falloir gérer les goulots d'étranglement qui en découlent.

Dans la seconde partie de ce travail, nous analysons plus en profondeur les problèmes que pose la sauvegarde en parallèle d'applications et nous proposons une approche de solution dans le but d'assurer les sauvegardes sans perte de performance du système global, tout en permettant une meilleure utilisation des ressources.

Deuxième partie

Proposition d'un ordonnancement des sauvegardes d'applications dans les environnements dynamiques

Chapitre 4

Analyse et prédiction de performance des opérations de sauvegarde

Sommaire

Dans le chapitre précédent, nous avons présenté l'importance de la sauvegarde en parallèle d'applications dans le contexte où les postes de travail libres d'un intranet peuvent être rebootés et intégrés dans une grappe virtuelle comme machine de calcul, et nous avons également évoqué les problèmes que pose cette approche. En effet dans une infrastructure de grappe virtuelle où plusieurs applications s'exécutant simultanément ont à se retirer pour libérer les ressources, sachant que le réseau et le disque du serveur sont partagés entres les différentes applications, on peut faire face à des contraintes d'utilisation des bandes passantes.

L'étude d'un système informatique part généralement de l'analyse fine de certains phénomènes expérimentaux liés au système pour construire un modèle permettant de prédire le comportement du système sous des hypothèses plus générales. La complexité que pose le problème de la sauvegarde en parallèle sur un même disque, comme nous allons le voir dans la suite de ce chapitre, a suscité très peu ou presque pas de travaux. Par conséquent il n'existe pas à notre connaissance une modélisation générale de ce problème.

Dans ce chapitre nous analysons donc les différents paramètres qui interviennent dans la sauvegarde en parallèle d'applications qui partagent le réseau et le disque du serveur. Nous montrons combien la pertinence de ces paramètres rend complexe le problème de la sauvegarde en parallèle d'applications. Sur la base d'un grand nombre d'expérimentations menées, nous faisons quelques propositions.

La complexité du problème de la sauvegarde en parallèle d'applications vers le disque du serveur est présentée à la section 4.1. La section 4.2 décrit l'environnement expérimental. La section 4.3 présente les résultats des expérimentations réalisées dans le cas des applications séquentielles et s'achève par la proposition d'une fonction bw qui donne la bande passante $bw(m, V)$ du système nécessaire pour la sauvegarde en parallèle de m applications de volume mémoire agrégé V sans que les performances du réseau et du disque ne se dégradent. Enfin la dernière section présente les expérimentations réalisées dans le cas des applications parallèles et relève la complexité d'établir un modèle de prédiction dans ce cas.

4.1 Complexité du système à modéliser

Cette section présente les différents paramètres matériels et logiciels qui interviennent lors de la sauvegarde en parallèle d'application vers un disque unique, ainsi que leur impact sur les performances du dispositif de sauvegarde (réseau et disque).

4.1.1 Chemin de sauvegarde

La sauvegarde sur le disque du serveur d'un processus s'exécutant sur un nœud de calcul utilise la totalité des composants réseaux qui relient ces deux machines. Le chemin de sauvegarde est la suite de composants traversés lors de cette sauvegarde. Ce chemin est constitué du bus mémoire, du bus PCI, de la carte réseau, du lien réseau, du commutateur, et à nouveau du lien réseau, puis de la carte réseau, du bus PCI et du disque, comme illustré à la figure 4.1.

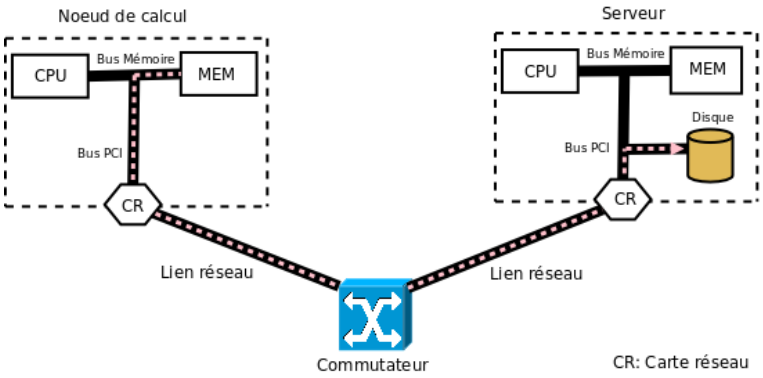

FIGURE 4.1 – Chemin d'une sauvegarde

Dans le cas de la sauvegarde à distance d'une seule application, en supposant qu'il n'y a pas d'autres communications vers le serveur, le lien réseau entre le commutateur et le serveur est exclusivement utilisé pour le transfert des données liées à cette sauvegarde. Cela n'est pas le cas lorsque plusieurs applications sont sauvegardées simultanément vers le même support.

4.1.2 Sauvegarde en parallèle

Dans le cas de la sauvegarde en parallèle de plusieurs applications vers le disque du serveur, la concurrence se produit au niveau de chacun des composants se situant sur plusieurs chemins de sauvegarde. La figure 4.2 montre dans l'ordre de numérotation, les composants qui peuvent être des facteurs pénalisant pour la sauvegarde en parallèle.

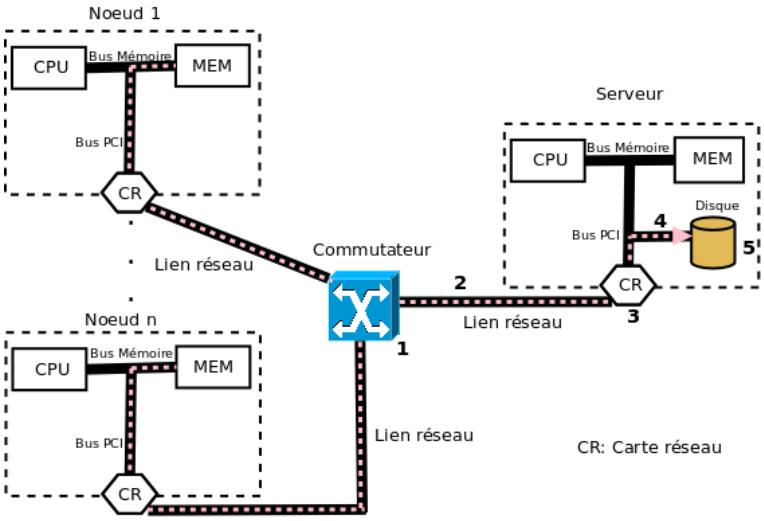

FIGURE 4.2 – Sauvegarde en parallèle

Influence des commutateurs

Pour accéder au disque, les points de reprise des processus à sauvegarder doivent simultanément emprunter le lien situé entre le commutateur et le serveur. La concurrence se produit alors au niveau de l'accès à ce lien. Ce scénario s'apparente aux communications concurrentes entrantes présentées dans [96]. Les commutateurs sont des composants qui interconnectent les nœuds à travers des liens réseaux. Dans le réseau Gigabit Ethernet, le commutateur est dans ce contexte le premier composant réseau pour la gestion de la concurrence. La plupart des commutateurs sont limités quant au flux qu'ils peuvent gérer. Certains processus peuvent alors être retardés, ce qui entraîne un surcoût du checkpointing. Cette situation peut être plus complexe encore si les points de reprise doivent traverser plusieurs commutateurs avant d'atteindre leur destination finale.

Influence des liens réseaux

Le second facteur qui peut influencer la performance du checkpointing est le lien réseau. Lorsque plusieurs points de reprise empruntent un même lien simultanément, la bande passante sur ce lien est partagée entre les différents nœuds d'où sont issus ces points de reprise. Cette diminution de la bande passante par nœud peut entraîner une augmentation de la latence du checkpointing, en fonction du volume de données à sauvegarder. Il serait alors intéressant de déterminer le bon ratio entre le nombre de processus à sauvegarder à travers le lien vers serveur et la bande passante nécessaire qui garantie une bonne performance du checkpointing.

Influence des cartes réseaux

Pour être acheminées vers le disque du serveur, les données transitent par la carte réseau. Les cartes réseaux disposent pour la plupart d'une mémoire tampon pour l'émission et pour la réception des données. Les données provenant du réseau sont provisoirement stockées sur le tampon de réception de la carte réseau avant d'être acheminées vers le disque à travers le bus PCI. Ce stockage provisoire a pour but de transformer les messages reçus au format du réseau en des données manipulables au sein de la machine. La mémoire tampon, en temps que composant partagé de la carte, peut donc être un facteur pénalisant dans la performance du checkpointing. En effet, en fonction du flux de données transmises vers la carte réseau du serveur, une partie des données peut être en file d'attente dans le réseau suivant la capacité de la mémoire tampon de la carte réseau, ce qui pourrait augmenter considérablement la latence du checkpointing.

Influence du bus PCI et du disque

Une fois que les données reçues du réseau sont traitées par la carte réseau, elles sont acheminées vers le disque du serveur via le bus PCI. Depuis leur mise au point par Intel en 1992, les bus PCI ont un débit sans cesse croissant. Initialement de 132 Mo/s, les débits des bus PCI atteignent aujourd'hui 4 Go/s. En fonction du débit du disque du serveur par rapport à celui du bus PCI, les données peuvent également connaître une pénalité quand à l'écriture sur disque.

Ceci peut entraîner un surcoût du checkpointing si le volume de données à écrire sur le disque est de grande taille.

4.1.3 Influence du système de fichier

Dans le contexte d'utilisation des ressources d'un intranet tel que nous l'avons adopté, étant entendu que toutes les opérations de lecture/écriture se font sur le disque du serveur, il est indiqué de virtualiser la proximité du disque. Ceci se fait à travers les systèmes de fichier réseau et permet aux nœuds de la grappe d'accéder au disque du serveur comme un accès local. Cependant les systèmes de fichier réseaux ont une limitation quant au nombre de clients qu'ils peuvent gérer à la fois. Par exemple NFS perd en performance [29, 97] avec la croissance du nombre de clients. Plus précisément les performances de NFS se dégradent pour des grappes ayant plus de 64 nœuds. Cela dit quand bien même le matériel ne constituerait pas un facteur de perte de performance du checkpointing, celle-ci pourrait se dégrader à cause du système de fichier. Même l'utilisation des systèmes de fichier parallèles tels que PVFS, qui offrent un plus grand passage à l'échelle que NFS, ne garantit pas toujours de bonnes performances du checkpointing, car ils ont également une limite.

4.1.4 Synthèse

Dans l'analyse que nous avons menée précédemment chaque composant (pris individuellement) du chemin de sauvegarde peut entraîner un surcoût du checkpointing lors de la sauvegarde en parallèle d'applications vers le disque du serveur. Leur mise en collaboration sur le chemin de sauvegarde peut avoir un impact plus fort encore sur la performance du checkpointing. Ceci montre la complexité de la détermination d'un modèle prédictif pour la sauvegarde d'applications en parallèle vers le disque du serveur sans perte de performance des mécanismes du checkpointing.

Au vu de la complexité qui se dégage de l'analyse technique que nous avons menée, nous proposons d'étudier dans les sections suivantes de manière expérimentale le problème de la sauvegarde en parallèle d'applications vers le disque du serveur.

Dans une analyse expérimentale préliminaire, nous avons envisagé d'étudier séparément l'effet de chaque composante du chemin de sauvegarde lors de la sauvegarde en parallèle d'applications. Par exemple nous avons isolé le disque pour tester son effet sur la sauvegarde en parallèle en local. Dans cette expérience préliminaire, un script était chargé d'exécuter plusieurs applications sur un même nœud et d'en assurer la sauvegarde en parallèle sur le disque local. Cette expérience a révélé un écroulement de la bande passante utilisée à partir d'un certain nombre de processus sauvegardés. Cependant il serait difficile d'assembler les informations issues des études isolées de chaque composante.

Fort de ce qui précède, nous considérons que la soumission des sauvegardes des applications est effectuée à travers une boîte noire qui encapsule toutes les composantes du chemin de sauvegarde et qui fournit en sortie les sauvegardes effectuées, comme le montre la figure 4.3.

Ceci permet de faire abstraction d'une analyse individuelle des composantes du chemin de sauvegarde.

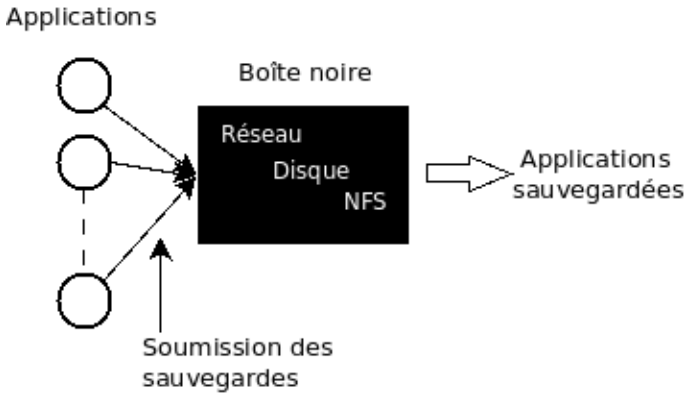

FIGURE 4.3 – Sauvegarde des applications à travers une boîte noire

Pour chaque application sauvegardée, nous considérons la bande passante spéciale utilisée pour la sauvegarde de l'application depuis la mémoire du nœud de calcul vers le disque du serveur. Cette bande passante est mesurée en faisant le rapport :

$$\frac{Volume\ de\ données\ sauvegardées\ (la\ taille\ du\ point\ de\ reprise)}{Temps\ de\ sauvegarde}$$

Cette précision sur la bande passante considérée est importante, car elle permet de ne pas faire la confusion avec la bande passante classique dont le débit est mesuré à partir d'une série de *ping-pong* de messages dont la taille varie, entre un émetteur et un récepteur dans un réseau.

Dans la suite nous considérons que la bande passante du système (celle de la boîte), c'est-à-dire la bande passante utilisée lors de la sauvegarde en parallèle des applications, est la bande passante agrégée des bandes passantes individuelles. La figure 4.4 illustre ces propos.

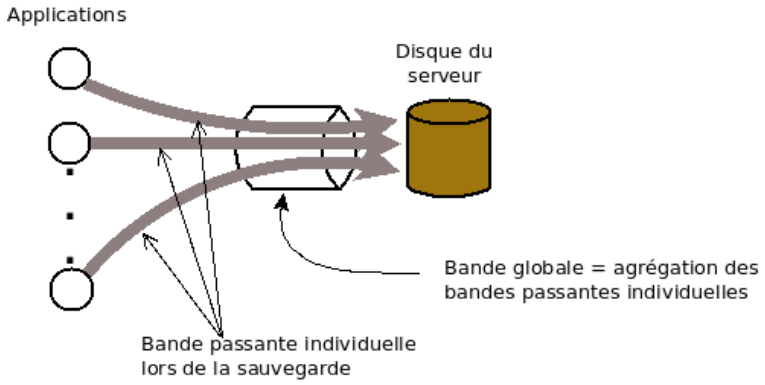

FIGURE 4.4 – Bande passante

Avant de présenter les expérimentations effectuées, nous commençons d'abord par décrire l'environnement expérimental dans la section qui suit.

4.2 Environnement expérimental

Cette section a pour but de présenter l'environnement matériel et les outils logiciels utilisés lors des expérimentations que nous avons menées. Nous y décrivons les applications utilisées et les objectifs expérimentaux.

4.2.1 Grille et grappes

Les expériences ont été menées sur certaines grappes de la grille Grid5000 [21], en l'occurence les grappes de Sophia, de Orsay et de Bordaeux. Ces grappes ont été choisies par rapport à l'architecture des nœuds que nous voulions identiques pour assurer des expériences reproductibles sur diverses grappes. Les nœuds utilisés sont des AMD Opteron bi-processeurs monocœur, ayant 2 GHz, 2 Go de RAM et 80 Go disque local (IDE-amd74xx). Nous avons créé une image contenant tous les outils nécessaires pour les expérimentations du checkpointing (lanceur de tâches en parallèle, les protocoles de système de fichiers partagés, les systèmes de checkpointing, etc.) Cette image a été déployée sur les nœuds réservés, l'un d'eux faisant office de serveur NFS. Ces nœuds sont interconnectés avec un commutateur Gigabit Ethernet et partagent le */home* (monté de façon synchrone) du serveur. Le montage du */home* de façon synchrone garantit que les données seront immédiatement écrites sur le disque, plutôt que de passer par un tampon, ce qui pourrait fausser les expériences.

4.2.2 Système de checkpointing

Pour la sauvegarde des applications séquentielles, nous avons choisi le système de checkpointing BLCR. Au moment où nous débutions cette thèse, BLCR était le système de checkpointing le plus largement utilisé du fait de sa compatibilité avec plusieurs architectures de machine. De plus BLCR est un système transparent à l'utilisateur, ce qui cadre bien avec nos objectifs, car nous ne manipulons que les exécutables des applications. Cette transparence convient également bien à des gestionnaires de ressources. Plus précisément nous avons travaillé avec la version BLCR-0.4.2 sous le système d'exploitation linux 2.6.12.1.

Pour la sauvegarde des applications parallèles nous avons utilisé LAM/MPI qui était au début de nos travaux la seule implémentaion des mécanismes de sauvegarde/reprise utilisant BLCR. De plus LAM/MPI fait du checkpointing coordonné, ce qui convient bien à notre contexte. Nous avons utilisé la version 7.1.4 de LAM/MPI.

4.2.3 Applications utilisées

BenchSeq : ce code séquentiel synthétique est un benchmark dont toutes les pages de la mémoire allouée sont sauvegardées. Les données à manipuler sont initialisées au sein de l'application. Nous l'avons codé pour évaluer le checkpointing d'applications séquentielles suivant plusieurs scénarios (applications de même taille, de taille différentes, etc.) et aussi pour confronter les résultats obtenus avec le code de calcul multigrilles qui utilise des données provenant de fichiers.

Code de calcul multigrilles : cette application séquentielle [98] est développée au sein du laboratoire ID-IMAG [1] pour la résolution des équations aux dérivées partielles. C'est une application dont la taille de la mémoire allouée croît considérablement lorsque l'on fait accroître le nombre de points sur les grilles.

BenchPar : ce code MPI synthétique est un benchmark que nous avons implémenté pour saturer le réseau. Les processus s'envoient et reçoivent des messages en permanence à travers l'appel de la procédure de communication collective *MPI_Alltoall* suivant le schéma présenté à la figure 4.5.

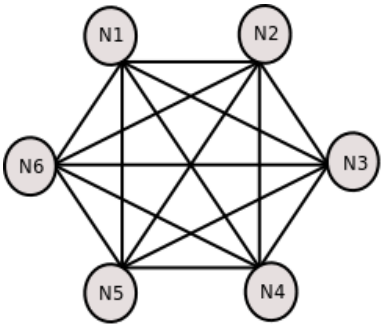

FIGURE 4.5 – Maillage complet dans un graphe d'échange de messages entre processus d'une application parallèle

Avec cette configuration, dans la phase de synchronisation au cours d'une sauvegarde, les canaux de communications dans le réseau sont saturés de messages.

cpi : c'est une application MPI pour le calcul en parallèle du nombre π. La précision de la valeur de π augmente avec le nombre d'itérations fixé. Plus le nombre d'itérations est grand, plus le temps d'exécution est long. Ce code est fourni avec la plupart des implémentation MPI. Nous avons modifié ce code de *cpi* pour l'adapter à nos tests. En l'occurrence nous y avons ajouté du code pour déployer l'application avec des précisions et des tailles en mémoire voulues. L'intérêt que nous portons pour ce code est qu'il n'y a pas de message en transit dans les canaux de communication au moment des sauvegardes, car les seules communications sont celles qui sont initiées par le processus maître au début (pour distribuer des tâches aux autres processus) et à la fin du calcul (pour collecter les résultats).

1. devenu LIG

4.2.4 Objectifs expérimentaux

Le but des expériences réalisées est l'étude des effets de la concurrence (pour l'utilisation des composantes du chemin de sauvegarde) sur le temps de sauvegarde en parallèle des applications. Il s'agit ici d'étudier la variabilité des temps de sauvegarde en fonction du nombre d'applications et de la taille agrégée de celles-ci. Cette variabilité des temps de sauvegarde est en forte corrélation avec la bande passante utilisée. Donc l'évolution de la bande passante pourrait traduire le niveau de performance des mécanismes du checkpointing. Pour les sauvegardes à distance, la bande passante considérée pour une application est celle du chemin de sauvegarde, c'est-à-dire depuis la mémoire sur un nœud jusqu'à l'écriture sur le disque du serveur. Ceci permet de faire abstraction du grand nombre de paramètres qui se trouve sur le chemin de sauvegarde, comme nous l'avons vu à la section 4.1.

Le contexte expérimental étant présenté, nous examinons dans les sections qui suivent les résultats obtenus des expérimentations menées et les propositions qui en découlent. Nous commençons par présenter le cas des applications séquentielles.

4.3 Cas des applications séquentielles

Dans cette section nous décrivons la méthodologie utilisée lors des expériences. Nous commençons par présenter le point de reprise généré par BLCR, car il est le point central des mesures effectuées. Nous analysons l'évolution de la bande passante utilisée en fonction du nombre d'applications sauvegardées et de leur taille agrégée. Cette analyse conduit à la proposition d'une estimation de la bande passante nécessaire pour la sauvegarde en parallèles d'applications sans perte de performance du système.

4.3.1 Méthodologie

A l'aide de l'outil *Taktuk* [99], nous avons exécuté en parallèles des applications séquentielles identiques indépendantes, une application par nœud, en faisant varier le nombre de nœuds n (de 1 à 30) et la taille v des applications. Au cours des sauvegardes, un script enregistre dans des fichiers de trace, le temps de sauvegarde et la taille des points de reprise. Pour chaque couple (n, v) plus de 100 mesures sont effectuées. Le temps de sauvegarde et la taille du point de reprise retenus pour chaque couple (n, v) sont obtenus en analysant la validité statistique (moyenne, écart-type) des mesures considérées. La moyenne a été le facteur clé.

4.3.2 Point de reprise généré par BLCR

A la section 3.5.3, nous avons vu qu'en fonction du système de checkpointing utilisé, le surcoût dû au checkpointing peut être élevé ou non. Pour analyser l'influence de BLCR lors de la sauvegarde, nous avons analysé le point de reprise généré par BLCR.

Par exemple, la figure 4.6 montre une divergence des tailles des points de reprise généré par BLCR pour la sauvegarde en local du code de calcul multigrilles et l'application synthétique BenchSeq, alors que ces deux applications occupent une même taille en mémoire. Cette disparité se traduit également sur les temps de sauvegarde correspondants, comme le montre la figure 4.7.

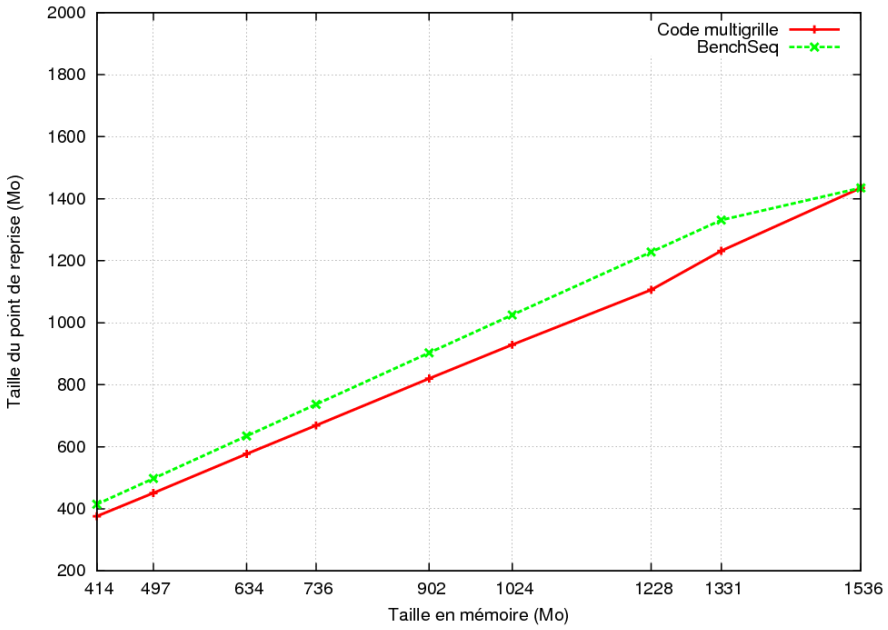

FIGURE 4.6 – Tailles du points de reprise générés par BLCR pour deux applications différentes occupant le même volume en mémoire

Nous rappelons que le code de calcul multigrilles utilise des données provenant de fichiers contrairement au code BenchSeq. Certaines pages du code de calcul multigrilles n'ont pas été sauvegardées. En fait BLCR effectue des optimisations lors de la sauvegarde. Par exemple les pages contenant des données provenant de fichiers ne sont pas enregistrées, du moment que les descripteurs et les pointeurs de fichiers le sont déjà. Ceci permet de réduire le volume de données à sauvegarder et par conséquent de réduire le temps de sauvegarde. Ceci justifie les disparités observées dans les figures 4.6 et 4.7.

Pour analyser le flux de données qui était écrit dans le point de reprise généré par BLCR, nous avons modifié BLCR, principalement le module *vmadump*, uniquement pour tracer l'écriture dans le point de reprise. Ceci nous a permis de constater que BLCR écrit essentiellement et alternativement des blocs de taille la longueur des entêtes des pages mémoires et la taille de ces pages (resp. 4 octects et 4 Ko pour notre Système d'Exploitation), pour les pages de la mémoire mappée du processus qui ont été modifiées au moins une fois, puis quelques informations relatives à l'architecture du nœud, aux signaux, etc. Nous avons émulé BLCR par un programme qui écrit des blocs dans un fichier comme le fait BLCR. Les figures 4.8 et 4.9 montrent les résultats obtenus pour une sauvegarde en local et à distance. Dans les deux cas,

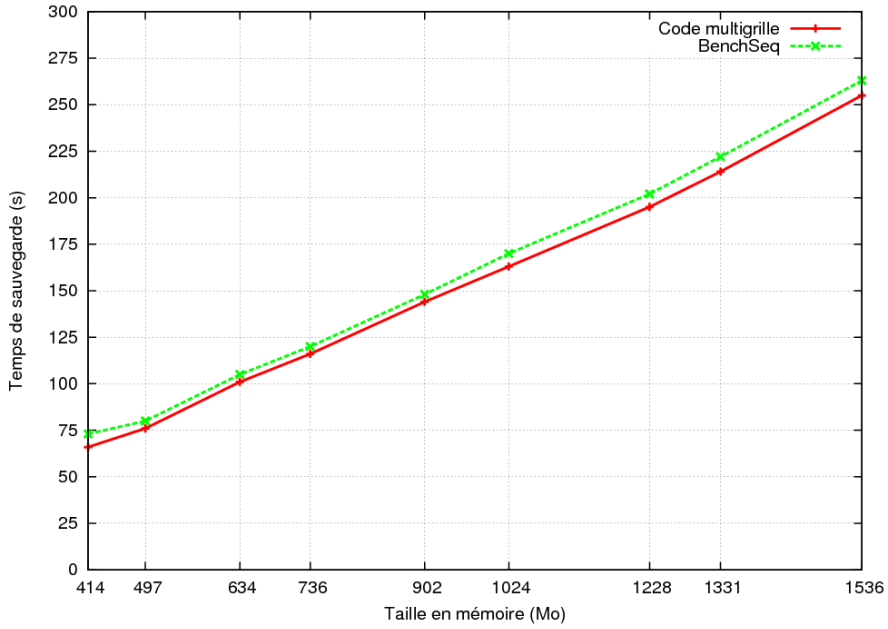

FIGURE 4.7 – Temps de sauvegarde correspondants

l'écart entre les temps de sauvegarde avec BLCR et par émulation est de l'ordre du centième de seconde. Ceci montre que le surcoût généré par BLCR pour traiter les pages à sauvegarder est négligeable.

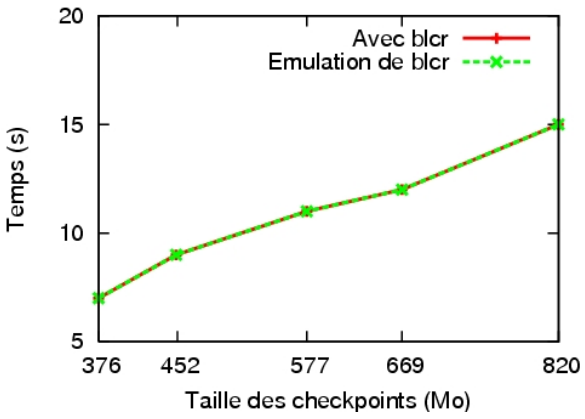

FIGURE 4.8 – Temps de sauvegarde du code de calcul multigrilles avec BLCR et par émulation de BLCR, en local

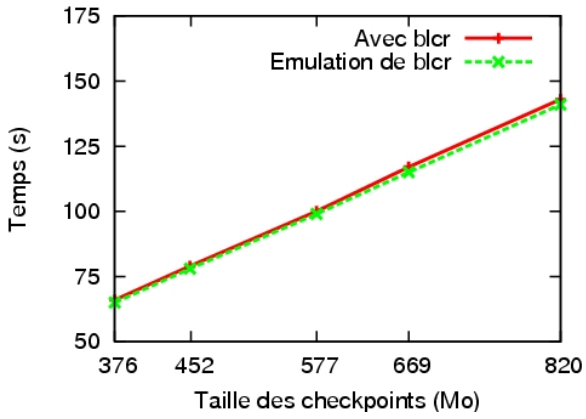

FIGURE 4.9 – Temps de sauvegarde du code de calcul multigrilles avec BLCR et par émulation de BLCR, à distance

Après l'analyse de l'impact de BLCR sur le point de reprise, observons l'évolution de la bande passante lorsque l'on fait varier le nombre de processus et leur taille en mémoire dans la

section qui suit.

4.3.3 Évolution de la bande passante

De nombreux tests ont été effectués lors de la sauvegarde en parallèle de plusieurs applications indépendantes en faisant varier le nombre de processus et leur taille en mémoire. Les tests ont été effectués avec des applications à sauvegarder en parallèle, de même taille. Des données enregistrées dans les fichiers de trace, nous avons pris comme estimation de la bande passante utilisée par chaque processus, le rapport entre la taille du point de reprise et le temps de sauvegarde. Des analyses, il ressort que dans l'environnement de test présenté, la bande passante était équitablement partagée entre les processus, même quand ceux-ci occupaient des volumes mémoire différents.

La figure 4.10 montre l'évolution de la bande passante utilisée en fonction du nombre de processus (un par nœud) pour différentes tailles du code de calcul multigrilles. Pour $i = 1, ..., 25$, les contextes des i premiers processus ont été sauvegardés et la bande passante expérimentale correspondante bw_i enregistrée. Les courbes de bw_i comme fonction de i sont illustrées dans la figure 4.10 pour différentes tailles en mémoire des processus.

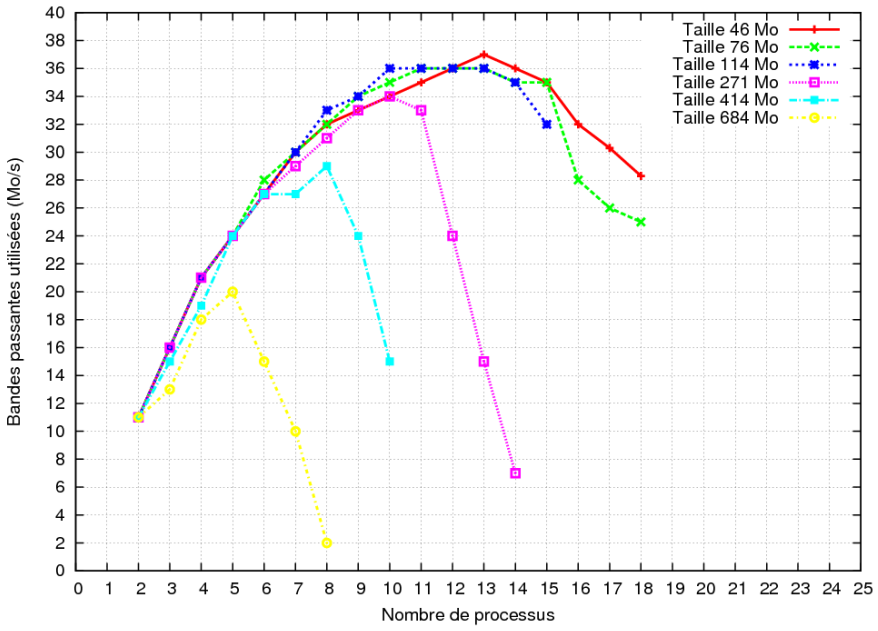

FIGURE 4.10 – Bande passante utilisée en fonction du nombre de processus selon la taille des applications

On constate que toutes ces courbes ont des allures paraboliques. Notons $(\bar{i}, bw_{\bar{i}})$ les pics de ces courbes. Avant le pic, la bande passante du réseau n'est pas entièrement utilisée et le disque tient la charge. Après le pic, la diminution de la bande passante utilisée indique une perte de performance du dispositif de sauvegarde. Ceci peut être dû à la forte latence générée par la

concurrence au niveau des différents composants du chemin de sauvegarde. Cependant, il est difficile de déterminer de façon précise l'impact du réseau ou du support de sauvegarde. Cela nous a conduit à adopter l'approche d'estimation présentée à la section suivante.

4.3.4 Proposition d'une estimation de la bande passante

Dans cette section, nous procédons par une analyse globale de l'évolution de la bande passante utilisée puis nous en déduisons une estimation.

Estimation de la bande passante utilisée

Étant donné que nous nous intéressons à la sauvegarde des applications sans perte de performance du dispositif de sauvegarde, nous ne nous intéressons qu'aux portions des courbes situées à gauche des pics. Plus précisément, les pics indiquent une utilisation optimale du dispositif de sauvegarde. Nous avons regroupé les mesures sur les pics dans le tableau 4.1.

Taille des applications (Mo)	$\bar{i}$	$bw_{\bar{i}}$
15	18	38,34
46	13	37,44
76	11	36,74
114	11	36,52
271	10	34,6
414	8	29,84
684	5	20,45

TABLE 4.1 – Récapitulatif des mesures effectuées pour les bandes passantes sur les pics

Nous avons observé l'évolution de la bande passante comme fonction de la taille agrégée des processus sur les pics comme le montre la courbe $\overline{V} \mapsto bw(\bar{i}, \overline{V})$ de la figure 4.11.

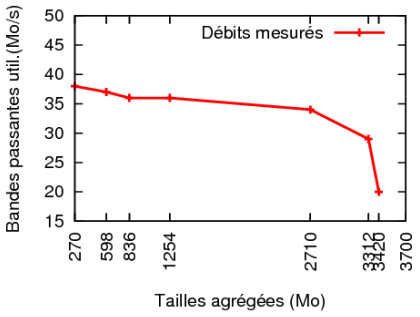

FIGURE 4.11 – Bande passante utilisée en fonction de V

Compte tenu des allures paraboliques des courbes $i \mapsto bw(i, V)$ (fig 4.10) et $\overline{V} \mapsto bw(\bar{i}, \overline{V})$ (fig 4.11), nous proposons d'interpoler la fonction $(m, V) \mapsto bw(m, V)$ par un polynôme de la forme :

$$bw(m, V) = \sum_{0 \leq i,j, \leq 2} a_{ij} m^i V^j \tag{4.1}$$

où m est le nombre de processus et V est la taille agrégée des processus.

Au vu de la concavité de la courbe de la figure 4.11, nous avons approximé la fonction $\overline{V} \mapsto bw(\bar{i}, \overline{V})$ par un polynôme du second degré et nous avons constaté que le coefficient du monôme de premier degré en V était quasiment nul. Ceci a donc réduit le nombre de paramètres de l'équation (4.1) qui devient :

$$bw(m, V) = am^2V^2 + bm^2 + cV^2 + dm + e \tag{4.2}$$

où les coefficients a, b, c, d, e sont déterminés par la méthode des moindres carrés comme suit :

Soit l'expression $y = bw(m, V)$ de la bande passante utilisée et soit à minimiser la fonction

$$\psi(a, b, c, d, e) = \sum_{i=1}^{r} \sum_{j=1}^{n_i} \left(y_{ij} - am_{ij}^2 V_j^2 - bm_{ij}^2 - cV_j^2 - dm_{ij} - e \right)^2$$

où
r est le nombre de tailles utilisées
n_i le nombre de mesures dans la taille v_i
m_{ij} le nombre de processus dans la taille v_i à chaque mesure j
y_{ij} les bandes passantes utilisées mesurées
V_i la taille agrégée obtenue à partir du pic de la courbe donnant la bande passante utilisée pour les applications de taille v_i.

Les coefficients a, b, c, d, e sont donc obtenus en résolvant les équations aux dérivées partielles :

$$\frac{\partial \psi}{\partial \omega} = 0, \omega \in \{a, b, c, d, e\}$$

Des mesures effectuées et dont les pics ont été récapitulés dans le tableau 4.1, on obtient :

$a = -0.0155$; $b = -0.169435$; $c = 0.0004$; $d = 5.027318$; $e = 3.753154$.

Avec ces coefficients, V est exprimée en Go dans la fonction $bw(m, V)$. La surface représentant bw est donnée dans la figure 4.12.

Notons b le minimum entre la bande passante du disque et la bande passante du réseau. Pour tout couple (m, V), on doit avoir $bw(m, V) < b$.
La fonction bw permet de calibrer le nombre d'applications qu'on peut sauvegarder simultanément sans perte de performance du système.

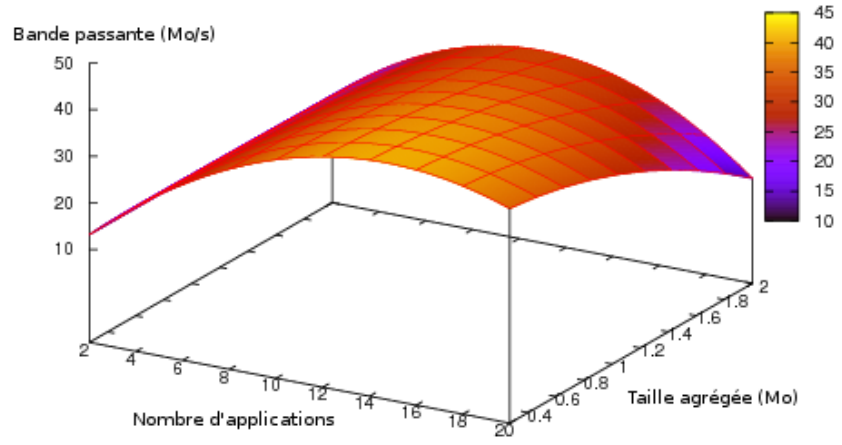

FIGURE 4.12 – Bande passante utilisée en fonction de m et V

4.3.5 Exemple

Considérons les trois ensembles D1, D2 et D3 représentant chacun les tailles en mémoire des 12 exécutions du code *BenchSeq* sur différentes tailles (c-à-d 12 applications de tailles différentes), l'unité de mesure des tailles étant le Mo :
D1=$\{105, 451, 135, 241, 329, 172, 211, 281, 117, 494, 113, 301\}$
D2=$\{20, 390, 57, 129, 425, 13, 330, 19, 211, 120, 493, 312\}$
D3=$\{412, 272, 320, 231, 455, 317, 401, 395, 429, 492, 201, 212\}$

Les courbes de bw_i mesuré comme fonction de i sont illustrées à la figure 4.13.

Les courbes de bw_i estimé comme fonction de i sont illustrées à la figure 4.14.

Les pics expérimentaux et estimés ($\bar{i}$, $bw_{\bar{i}}$) pour ces trois exemples sont présentés dans le tableau 4.2. On peut constater que ces estimations sont très proches des résultats expérimentaux.

	D1	**D2**	**D3**
$bw_{\bar{i}}$ mesuré (Mo/s)	31.20	34.03	28.77
$bw_{\bar{i}}$ estimé (Mo/s)	30.04	32.53	26.24
$\bar{i}$ mesuré	9	11	8
$\bar{i}$ estimé	9	10	7

TABLE 4.2 – Pics mesurés et estimés

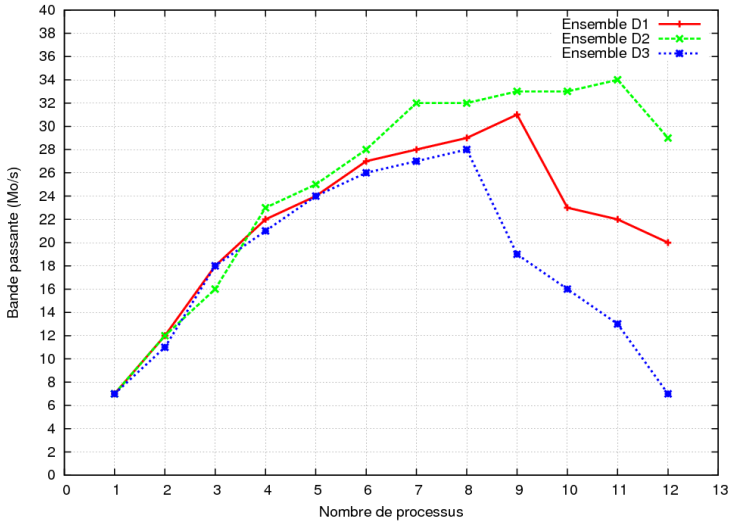

FIGURE 4.13 – Bandes passantes expérimentales

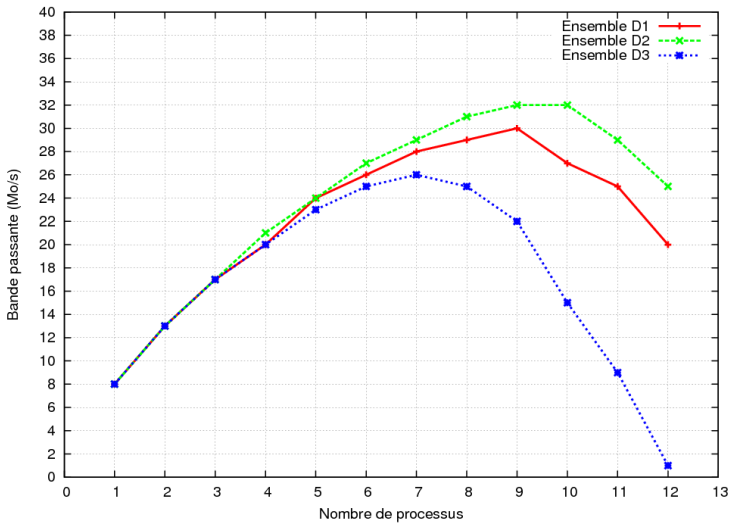

FIGURE 4.14 – Bandes passantes estimées

Pour mieux apprécier la validité de la fonction bw, il est important d'observer l'erreur qui s'en dégage.

Erreur d'estimation

Pour évaluer la validité de la fonction bw, une comparaison est faite entre les bandes passantes mesurées (y_{ij}) et la bande passante estimé $bw_{ij} = bw(m_{ij}, m_{ij} * v_i)$, pour les valeurs obtenues jusqu'aux pics. Les valeurs après les pics ne présentent aucun intérêt car elles in-

diquent des baisses de performance du dispositif de sauvegarde.

Posons $\overline{n_i}$ ($\overline{n_i} \leq n_i$)le nombre de mesures jusqu'au pic pour la taille v_i, $i = 1, ..., r$. L'erreur globale est obtenue par la métrique suivante exprimée en pourcentage :

$$E_{globale} = \frac{1}{\sum_{i=1}^{r} \overline{n_i}} \sum_{i=1}^{r} \sum_{j=1}^{\overline{n_i}} |\frac{y_{ij} - b_{ij}}{y_{ij}}| * 100$$

Cette moyenne donne une vision globale des précisions obtenues à chaque comparaison. Un code simple a permis d'obtenir la valeur de $E_{globale} = 4,02\%$ à partir des mesures effectuées et du calcul de bw. Cela montre que l'estimation de la bande passante utilisée est efficace.

Dans la section qui suit, nous présentons l'analyse effectuée dans le cas de la sauvegarde d'applications parallèles.

4.4 Cas des applications parallèles

Dans cette section, nous montrons la complexité de la détermination d'un modèle de prédiction du temps de sauvegarde des applications parallèles.

4.4.1 Surcoût induit par la synchronisation

Nous rappelons que nous nous intéressons à la sauvegarde coordonnée dans le cas des applications parallèles. Lors de la sauvegarde d'une application parallèle, les processus synchronisent dans un premier temps leurs états, pour déterminer un état global consistant avant d'être individuellement et indépendamment sauvegardés. Le but de cette section est de présenter le surcoût induit par la phase de synchronisation lors du checkpointing d'une application parallèle, le système de checkpointing utilisé étant LAM/MPI.

Le tableau 4.3 présente les résultats des tests obtenus à partir du checkpointing de l'application parallèle *cpi* dont il n'y a aucun message dans les canaux de communication au moment du checkpointing et les applications séquentielles dont le volume agrégé des données à sauvegardé est le même que celui de l'application parallèle. Pour ces tests les processus de l'application parallèles ont la même taille que ceux des applications séquentielles correspondantes. Pour effectuer des comparaisons raisonnables, nous avons travaillé des nombres de processus dont la sauvegarde n'induit pas des pertes de performance du système global.

Les mesures sont regroupées suivant trois types de sauvegardes :
- la sauvegarde d'une application séquentielle qui n'induit aucune concurrence. Le volume de données à sauvegarder est égal au volume agrégé de plusieurs applications séquentielles indépendantes et au volume agrégé des processus d'une application parallèle
- la sauvegarde de plusieurs applications séquentielle qui induit une concurrence pour l'utilisation du réseau et du disque.
- la sauvegarde d'une application parallèle qui passe par une phase de synchronisation avant la sauvegarde individuelle de chacun de ses processus.

		Taille agrégée des processus S (Mo) et temps de checkpoint T (s)									
	Nombre processus	S——T		S——T		S——T		S——T		S——T	
App. séq. unique	1	20	**4**	80	**15**	**100**	**18**	400	**73**	600	**108**
Plusieurs applications séquentielles	2	2 x 10	**2**	2 x 40	**8**	2 x 50	**8**	2 x 200	**38**	2 x 300	**59**
	4			4 x 20	**4**	4 x 25	**5**	4 x 50	**20**	4 x 150	**27**
	5					5 x 20	**4**	5 x 80	**17**	5x 120	**23**
	8			8 x 10	**2**			8 x 50	**13**	8 x 75	**19**
	10					10 x 10	**3**	10 x 40	**12**	10 x 60	**18**
	15									15 x 40	**15**
Application parallèle	2	2 x 10	**4**	2 x 40	**32**	2 x 50	**37**	2 x 200	**147**	2 x 300	**211**
	4			4 x 20	**31**	4 x 25	**36**	4 x 50	**147**	4 x 150	**212**
	5					5 x 20	**37**	5 x 80	**148**	5x 120	**209**
	8			8 x 10	**30**			8 x 50	**147**	8 x 75	**212**
	10					10 x 10	**37**	10 x 40	**149**	10 x 60	**211**
	15									15 x 40	**213**

TABLE 4.3 – Comparaison entre les temps de sauvegarde d'une application parallèle et de plusieurs applications séquentielles

Pour faciliter la lecture des données du tableau 4.3, nous précisons que la taille agrégée de m processus est notée m x s où s est la taille de chaque processus.

Ces mesures révèlent d'importantes informations :

- la sauvegarde en parallèle d'un bloc de données est plus efficace que la sauvegarde séquentielle de tout le bloc, lorsqu'il n'y a pas de perte de performance du dispositif de sauvegarde. On peut le constater en comparant les temps de sauvegarde d'une application séquentielle à ceux de plusieurs applications séquentielles indépendantes dont le volume agrégé est le même que celui de l'application séquentielle.
- pour une application parallèle de m processus de taille chacun s et de volume agrégé $S = ms$, dont les processus ne communiquent pas au moment où le checkpointing est initié, le temps de sauvegarde est presque équivalent à celui de la même application ayant $m' = \frac{m}{p}$ processus de taille chacun $s' = ps$ et de taille agrégé $S' = m's'$. Ceci montre que lorsqu'il n'y a pas de message dans les canaux de communications au moment du checkpointing, le temps du checkpointing est uniquement dépendant du volume de données à sauvegarder et de la phase de synchronisation. On le constate bien sur le tableau 4.3.
- on constate que le temps de sauvegarde de l'application parallèle est d'au moins le double que celui de l'application séquentielle unique et très largement au dessus du temps de sauvegarde des applications séquentielles indépendantes ayant le même volume agrégé. Cela traduit le fait que la phase de synchronisation induit un surcoût considérable sur le temps du checkpointing.

Lorsque les processus d'une application parallèle communiquent par échange de messages au moment où est initié le checkpointing, le surcoût dû à la sauvegarde est encore plus fort, comme le montre la section suivante.

4.4.2 Impact de la présence de messages dans les canaux de communication lors de la sauvegarde d'une application parallèle

Pour mesurer l'impact de la présence des messages dans les canaux de communications sur le temps de sauvegarde d'une application parallèle, nous avons utilisé l'application *BenchPar* présentée à la section 4.2.3. Le fait que chaque processus envoie des messages à tous les autres et reçois également de tous de façon permanente permet de garantir de la présence des messages dans les canaux de communication au moment où le checkpointing est initié.

Nous avons fait varier la taille des messages échangés par les processus, ainsi que le nombre de processus. Les résultats de ces tests sont présentés dans figure 4.15. Pour ces tests, la taille de chaque processus a été de 10 Mo.

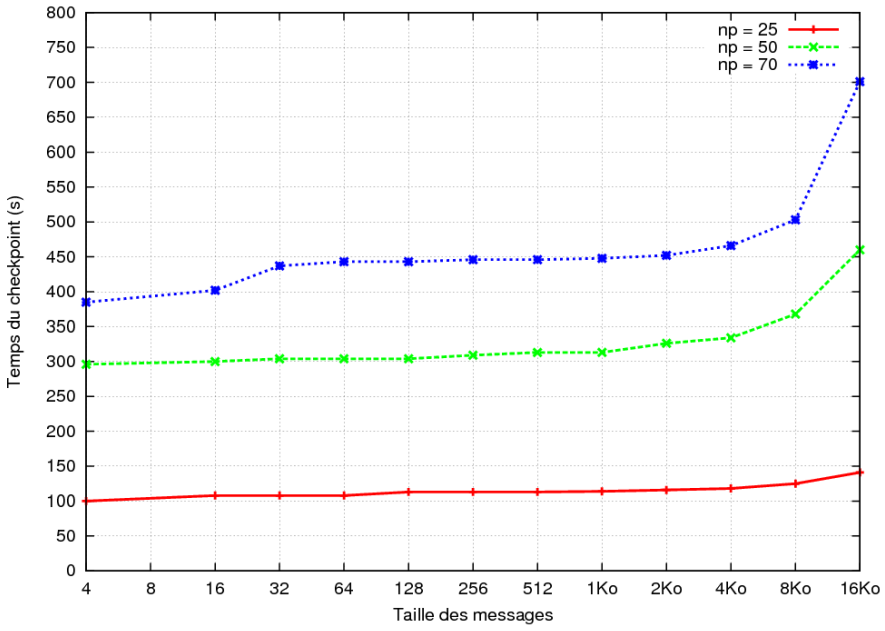

FIGURE 4.15 – Comparaison des temps de sauvegarde d'une application parallèle pour différentes tailles des messages dans les canaux de communication

On constate que le temps de sauvegarde croît en fonction de la taille des messages, et que cette croissance est plus accentuée dans le cas où le nombre de processus devient grand. Ceci montre que la présence de messages dans les canaux de communication induit une augmentation du temps du checkpointing, ce qui peut entraîner de fortes pénalités au niveau des performances du checkpointing. Il convient de noter que ces résultats peuvent être fortement dépendant de la politique de sauvegarde de LAM/MPI.

4.4.3 Complexité de la prédiction du temps de sauvegarde d'une application parallèle

L'estimation du temps de sauvegarde d'une application n'est pas aussi simple que ce que nous avons présenté précédemment, car la présence de messages dans les canaux de communications peut faire varier le temps de sauvegarde de façon drastique. Par exemple la figure 4.16 compare les temps de sauvegarde de l'application *BenchSeq* utilisé avec des processus qui s'échangent en permanence des messages de 16 ko et ceux de l'application *cpi* où il n'y a pas de communication au moment de la sauvegarde. Pour ces tests également, la taille de chaque processus a été de 10 Mo.

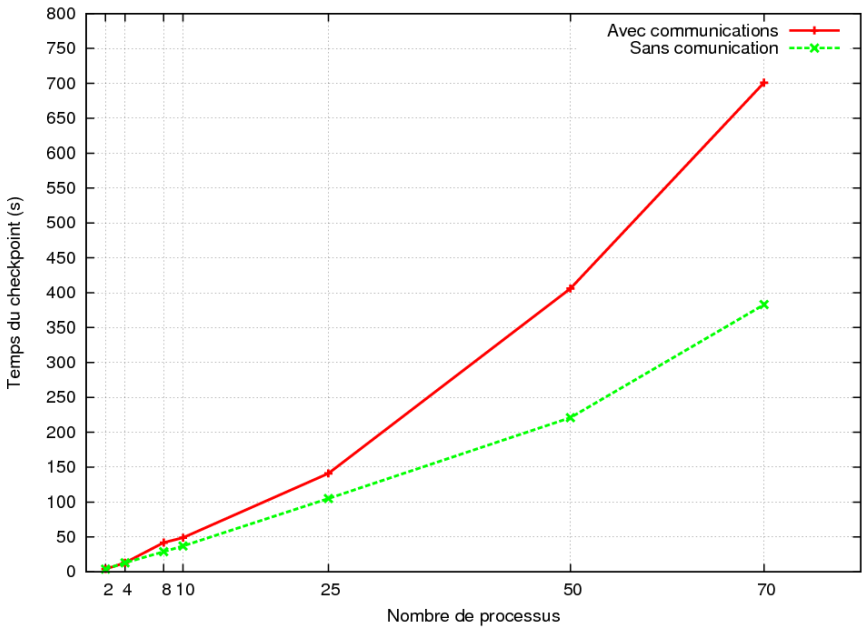

FIGURE 4.16 – Comparaison des temps de sauvegarde de deux applications parallèles dont l'une a des messages dans les canaux de communication

On constate que l'écart entre le temps de sauvegarde avec communication et sans communication devient important lorsque le nombre de processus croît.

En général il est très difficile de prédire la présence de messages dans les canaux de communication lors de la sauvegarde d'une application parallèle avec une implémentation du checkpointing de niveau système. Les expériences que nous venons de présenter montrent que même si le checkpointing est fait au niveau applicatif, où le programmeur a une parfaite connaissance du flux de données manipulées, la taille des messages échangés et le nombre de processus en communication rendent complexe le problème de la prédiction du temps de sauvegarde d'une application parallèle.

4.5 Conclusion

Dans ce chapitre nous avons analysé l'impact de certains composants matériels et logiciels sur les performances des opérations de sauvegarde en parallèle d'applications qui partagent le réseau et le disque du serveur. Il ressort de l'analyse menée que les processus sont en concurrence à différents niveaux du chemin de sauvegarde, mais il est difficile de déterminer de façon précise et individuelle l'action de chaque paramètre ayant un impact sur les performances du système.

Sur la base d'un grand nombre d'expérimentations, nous avons proposé une fonction bw qui donne la bande passante $bw(m, V)$ du système nécessaire pour la sauvegarde en parallèle de m applications séquentielles de volume mémoire agrégé V. Ceci nous a amené à proposer pour bw un polynôme du second degré par rapport à chacune des variables m et V.

Pour le cas des applications parallèles, nous avons montré qu'il est très difficile d'établir un modèle prédictif pour la sauvegarde en s'inspirant des expériences. En effet en considérant seulement le cas de la sauvegarde d'une seule application parallèle ayant r processus, nous avons montré que le temps de sauvegarde varie en fonction de la taille des messages dans les canaux de communication et de la taille des messages échangés lors de la phase de synchronisation. Ceci rend très complexe l'étude d'une estimation de la bande passante utilisée, car on ne peut pas prévoir l'existence de messages en transit au moment du checkpointing, encore moins la taille de ces messages. Le problème est plus compliqué encore lorsque qu'il s'agit de la sauvegarde de plusieurs applications parallèles.
Cette complexité explique le choix que nous avons fait de ne traiter que le cas des applications séquentielles indépendantes.

Comme nous l'avons vu, la fonction bw permet de calibrer le nombre d'applications que l'on peut sauvegarder sans perte de performance du système global. Cela suppose que lors de la libération des ressources de l'intranet avant une date $date\_fin$ fixée, les processus inachevés pourront être sauvegardés sur le disque du serveur. Cependant on n'est pas sûr de pouvoir les sauvegarder toutes dans le délai imparti.

Dans le chapitre qui suit, nous proposons un modèle d'ordonnancement des sauvegardes des applications lorsque l'on ne dispose que d'un delai T avant la libération complète des ressources de l'intranet. La fonction bw y est utilisée pour estimer le temps de sauvegarde de chacune des m applications, dans un contexte où la bande passante $b(m, V)$ est équitablement partagée entre les m tâches.

Chapitre 5

Ordonnancement des sauvegardes

Sommaire

L'approche d'utilisation des ressources libres d'un intranet adoptée dans le cadre de cette thèse consiste à basculer les postes de travail acquis du mode machine de bureau vers le mode machine de calcul dans un environnement dynamique de type grappe virtuelle, pour effectuer du calcul haute performance. Dans une grappe virtuelle, la volatilité des ressources est l'une des contraintes à prendre en compte. En effet, généralement, lorsque les ressources disponibles de l'intranet d'une entreprise sont utilisées comme une grappe virtuelle pour les calculs scientifiques, les périodes d'inactivité ne permettent pas toujours d'exécuter les calculs qui leur sont alloués jusqu'à leur terme.

Lorsque des postes de travail exécutant chacun une des applications $\{A_1, A_2, ..., A_n\}$ doivent être libérés dans un délai T, il est nécessaire de sauvegarder le contexte d'exécution de ces applications pour une éventuelle reprise. Toutefois, compte tenue des contraintes liées à la concurrence pour l'utilisation du réseau et du disque, il n'est pas toujours possible de sauvegarder toutes les applications dans le délai T imposé pour libérer les postes de travail. Une bonne approche serait de sauvegarder une collection de tâches qui maximisent la consommation des ressources. Nous supposons ici que la consommation de ressources correspondant à une tâche est proportionnelle au temps de calcul non encore sauvegardé de cette tâche. Nous sommes donc confrontés à un problème d'optimisation qui consiste à ordonnancer dans le délai T, une collection de tâches $\{A_{i_1}, A_{i_2}, ..., A_{i_k}\}$ qui maximisent la somme $p_{i_1} + p_{i_2} + ... + p_{i_k}$ des temps de calcul non encore sauvegardés des tâches correspondantes, sous contraintes des bandes passantes réseau et disque.

Comme nous l'avons vu à la section 3.5.1, le temps de sauvegarde dépend fortement des tailles en mémoire $s_{i_1}, s_{i_2}, ..., s_{i_k}$ des processus sauvegardés. En effet, le volume de mémoire occupé par une application est l'élément le plus important dans la création du point de reprise. Il est raisonnable de faire une prédiction du temps de sauvegarde, en supposant que toutes les pages mémoire d'une application doivent être sauvegardées, même si dans la plupart des situations, ce n'est pas vrai. Cette prévision peut être améliorée si l'on sait à l'avance la quantité exacte de données à sauvegarder. Malheureusement, une telle prédiction nécessite une grande intrusion dans le processus en cours d'exécution, ce qui peut sérieusement affecter les performances des applications. Elle n'est donc pas utilisée dans notre contexte.

Pour une bonne compréhension de la suite du travail, nous rappelons deux considérations fondamentales que nous avons faites au chapitre 2 :

- les applications utilisées résident entièrement en mémoire. Cela permet d'éviter de sévères pénalités sur la performance des applications et du système de sauvegarde. En effet, comme nous l'avons vu à la section 2.3.2, les applications dont le volume de données ne peut pas tenir en mémoire sont obligées de faire du *swap* entre la mémoire et une partition du disque où est stockée une partie des données. Cela induit très souvent un surcoût sur les performances du système évalué.
- Les points de reprise sont sauvegardés sur le disque du serveur, bien que cette sauvegarde soit sujette à des contraintes (figure 5.1). Cette considération permet d'éviter toute interférence avec l'espace utilisateur et permet également de palier au fait que le poste de travail pourrait ne plus être disponible à la prochaine exploitation des ressources libres de l'intranet.

Dans ce chapitre nous proposons une approche pour l'ordonnancement sous ces contraintes,

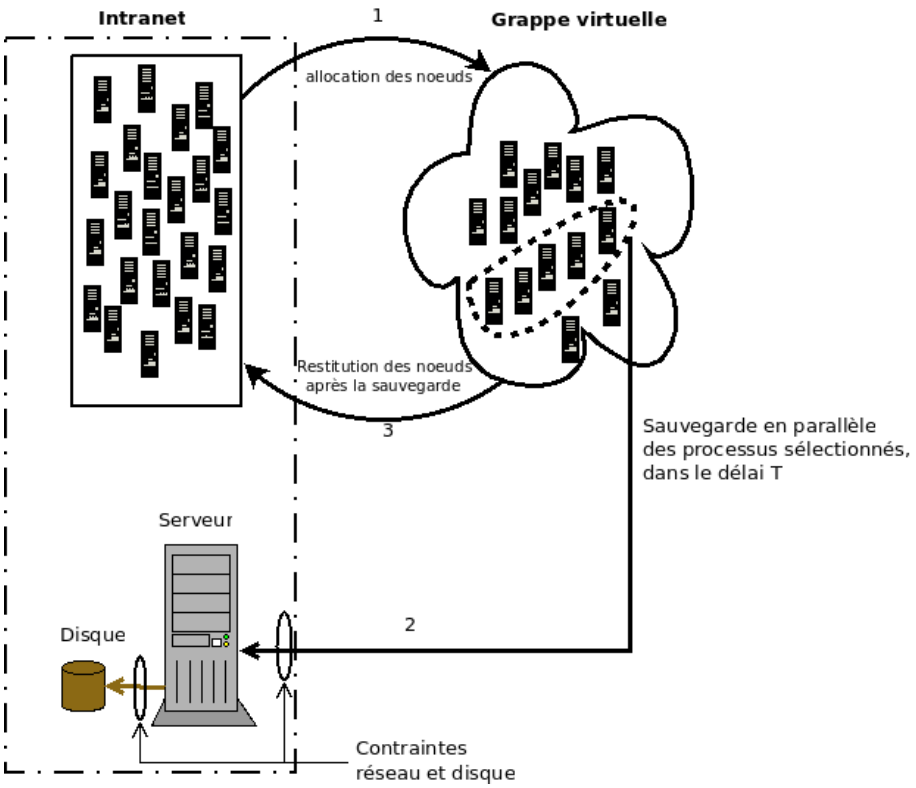

FIGURE 5.1 – Sauvegarde en parallèle d'applications avec contrainte dans une grappe virtuelle

des sauvegardes d'applications dans les environnements dynamiques comme les grappes virtuelles. Nous donnons une formulation du problème d'ordonnancement des sauvegardes et nous proposons un schéma d'approximation de la solution basé sur une toute nouvelle version du problème du sac-à-dos 0/1 où la contrainte intègre la fonction bw présentée au chapitre 4. Nous démontrons que le problème d'ordonnancement formulé est NP-Complet.

L'approche que nous proposons ici pour résoudre le problème d'ordonnancement des sauvegardes avec contrainte temporelle détermine dans un premier temps la bande passante $bw(m, V)$ nécessaire pour la sauvegarde en parallèle de m applications de volume mémoire agrégé V. Ensuite nous introduisons une boucle dans laquelle les processus sélectionnés pour le checkpointing maximisent la consommation des ressources (et implicitement $bw(m, V)$) tout en tenant compte de la contrainte temporelle T. A chaque boucle, pour la sélection des processus pour le checkpointing, nous utilisons une heuristique basée sur l'algorithme approché du problème du sac-à-dos proposé par Sartaj Sahni [8] qui combine une approche semi-énumérative à une stratégie gloutonne.

Afin de présenter plus en détail la proposition d'une stratégie d'ordonnancement des sauvegardes, nous commençons par faire un rappel du problème du sac-à-dos 0/1 et nous présentons l'algorithme approché de Sartaj Sahni.

5.1 Algorithmes approchés du problème du sac-à-dos 0/1

Le problème du sac-à-dos 0/1 consiste à trouver une combinaison de différents objets que l'on choisit pour son sac-à-dos de capacité limité, combinaison dans laquelle la valeur totale de tous les objets choisis est maximisée. Le problème du sac-à-dos 0/1 est le plus important de la famille des problèmes du sac-à-dos et l'un des problèmes de programmation discrète les plus amplement étudiés. Son intérêt provient de trois facteurs fondamentaux [100] : (i) il peut être vu comme le plus simple des problèmes de programmation linéaire en nombre entiers ; (ii) il se présente comme un sous-problème de plusieurs problèmes plus complexes ; (iii) Il permet de modéliser un grand nombre de problèmes concrets.

Le problème du sac-à-dos 0/1 prend comme paramètres d'entrée deux ensembles de r entiers positifs $P = \{p_1, p_2, ..., p_r\}$, $S = \{s_1, s_2, ..., s_r\}$ et un entier M. Les p_i peuvent être interprétés comme étant les gains associés aux objets $i = 1, 2, ..., r$ et s_i peut être vu comme le poids ou la taille de l'objet i. M est la taille du sac-à-dos. Si l'objet i est mis dans le sac, il génère un gain p_i et l'objet i occupe l'espace s_i. Le problème du sac-à-dos 0/1 est formulé comme suit :

$$\begin{cases} maximiser \sum_{i \in I} p_i \delta_i \\ sous\ contrainte \sum s_i \delta_i \leq M \end{cases} \tag{5.1}$$

où $\delta_i = 1$ si l'objet i est sélectionné et $\delta_i = 0$ sinon.

Trois principales stratégies gloutonnes peuvent être utilisées pour résoudre (5.1) :
- remplir le sac-à-dos dans l'ordre des densités décroissantes (p_i/s_i),
- remplir le sac-à-dos dans l'ordre des gains décroissants (p_i),
- remplir le sac-à-dos dans l'ordre des poids décroissants (s_i).

Le problème du sac-à-dos 0/1 a été largement étudié dans la littérature au cours des cinq dernières décennies, en fonction des avancées théoriques de l'optimisation combinatoire. Nous donnons dans les paragraphes suivants une chronologie non exhaustive des travaux sur le problème du sac-à-dos.

En 1954, la théorie de programmation dynamique de Bellman [101] a produit des algorithmes exacts pour résoudre le problème du sac-à-dos 0/1. En 1957 Dantzig [102] a développé une méthode efficace pour déterminer la solution de la relaxation continue du problème du sac-à-dos 0/1, en fixant une borne supérieure à la fonction objectif. Cette méthode a été utilisée par la suite dans presque tous les travaux portant sur le problème du sac-à-dos. Notons que la relaxation continue est une méthode utilisée pour obtenir des informations du problème discret initial. Elle consiste à interpréter de façon continue un problème combinatoire ou discret.

Kolesar a introduit en 1967 dans [103] le premier algorithme par séparation et évaluation (*branch-and-bound*) pour le problème du sac à dos. Cette technique est implémentée pour trouver les solutions optimales des problèmes d'optimisation. Elle est basée sur l'élimination en masse des candidats. La technique *branch-and-bound* a été davantage développée dans les années 1970 car s'étant avérée être la seule méthode capable de résoudre des problèmes ayant un grand nombre de variables.

En 1973, Ingargiola et Kosh [104] présente la première procédure de réduction, un algorithme qui réduit de manière significative le nombre de variables du problème initial. En 1974 Horowitz et Sahni [105] ont proposé un algorithme utilisant l'approche *branch-and-bound* pour rechercher les candidats à éliminer suivant un partitionnement de l'ensemble des candidats. cet algorithme bien que meilleur que ceux existants à cette époque présentait le désavantage d'être exponentiel.

Le premier schéma d'approximation en temps polynômial pour le problème du sac-à-dos 0/1 a été proposé en 1975 par Sartaj Sahni dans [8]. En 1977, Martello et Toth ont proposé la première borne supérieure dominant la valeur de la relaxation continue. Martello et Toth ont poursuivi leur travaux et ont proposé en 1988 dans [106] un nouvel algorithme approché pour la résolution du problème du sac-à-dos 0/1 basé sur la détermination d'un sous-ensemble approprié d'items dont la solution permet d'obtenir celle du problème original à partir d'une heuristique.

Les années 1990 ont été marquées par de nouvelles approches qui utilisent des algorithmes parallèles [107] pour la résolution du problème du sac-à-dos. Une approche basée sur l'utilisation des réseaux de neurones a été introduite dans [108].

La plupart des développements récents comme [109], [110], [111] utilisent des algorithmes génétiques qui permettent d'obtenir une solution approchée du problème d'optimisation en un temps acceptable. Cependant, dans la plupart des cas, les évolutions récentes sont multi-objectifs ou multidimensionnelles.

Il a été démontré dans [112] que le problème du sac-à-dos est NP-complet, c'est-à-dire que s'il existe un algorithme qui s'exécute en temps polynômial pour le problème du sac-à-dos, alors on peut trouver des solutions en temps polynômial pour une grande variété de problèmes pour lesquels, actuellement, il n'existe aucune solution connue en temps polynômial.

Dans [100], Martello et Toth ont comparé expérimentalement leur algorithme approché pour le problème du sac-à-dos 0/1 proposé dans [106] avec l'algorithme approché de Sartaj [8]. De leur analyse, il ressort que pour de petites valeurs du nombre d'objets, l'algorithme de Sartaj peut produire de meilleures approximations. Puisque dans notre contexte, nous nous attendons à avoir une meilleure consommation agrégée de ressources, et puisque nous travaillons sur de petites valeurs du nombre d'objets, l'algorithme approché de Sartaj, bien que ancien, apparaît comme la meilleure approche pour nous.

Dans [8] l'agorithme approché suivant (*SS-Glouton*) avec le paramètre k_0 a été présenté : *Considérer toutes les combinaisons* $\{i_1, i_2, ..., i_k\}$ *de k objets,* $k \leq k_0$*, ayant une taille totale d'au plus M (la capacité du sac). Pour chaque combinaison, construire un candidat en le complétant avec une stratégie gloutonne. La solution produite par cet algorithme est le meilleur candidat.*

Ce condensé permet d'avoir une bonne compréhension de la suite du travail qui commence par la description du problème dans la section qui suit.

5.2 Description du problème

On considère n applications séquentielles indépendantes en cours d'exécution dans une grappe virtuelle constituée des postes de travail d'un intranet, une application par nœud. On suppose que les postes de travail doivent être libéré dans un délai T pour la restitution des ressources. On suppose également que les points de reprise doivent être sauvegardés sur le disque du serveur de l'intranet. Pour des raisons d'utilisation efficace des ressources, les sauvegardes d'applications sont faites en parallèle. Cependant, comme nous l'avons déjà vu, compte tenu des contraintes liées à la concurrence pour l'utilisation du réseau et du disque, on n'est pas sûr de sauvegarder toutes les applications dans le délai T.

Notons p_i la valeur représentant la consommation de ressources par l'application i, c'est-à-dire le temps de calcul non encore sauvegardé de l'application i. Le problème à résoudre consiste à sélectionner un sous-ensemble $\{i_1, i_2, ..., i_r\}$ d'applications qui peuvent être sauvegardées dans le délai T et tel que la somme $p_{i_1} + p_{i_2} + ... + p_{i_r}$ soit maximale. Dans cette formulation, les p_i' sont de grands entiers car nous supposons que les applications considérées ont un long temps d'exécution.

Avec ces considérations, nous sommes confronté au problème suivant :

$$\begin{cases} maximiser \sum_{i \in E} p_i \delta_i \\ sous\ contrainte\ temps - de - sauvegarde\{Tâche_i, i \in E\} \leq T \end{cases} \tag{5.2}$$

avec $\delta_i \in \{0, 1\}$

Il n'existe pas de formule simple et globale pour la contrainte de ce problème d'optimisation. En effet, étant donné que les contraintes de bandes passantes réseau et disque ne permettent pas toujours d'engager toutes les sauvegardes à la fois, toute approche raisonnable devra d'abord sélectionner une série de tâches pour démarrer le processus de sauvegarde. Ensuite, à la fin de la sauvegarde des points de reprise d'un groupe de ces tâches, le système choisira de nouvelles tâches candidates qui seront incluses dans l'ensemble des tâches en cours de sauvegarde. Cela conduit à une boucle.

Dans la section suivante, nous proposons un exemple d'une telle boucle qui produit une solution approchée de ce problème d'optimisation complexe.

5.3 Schéma d'approximation

Dans cette section, nous présentons une formulation détaillé du problème (5.2). Nous démontrons que ce problème est NP-Complet et nous proposons une boucle qui produit une solution approchée de ce problème.

5.3.1 Formulation détaillée du problème

Nous considérons les notations suivantes qui seront utilisées dans la suite de ce chapitre :
$E = \{1, ..., n\}$ est l'ensemble des indices des n applications à sauvegarder.
T est le délai imposé.
$P = \{p_1, ..., p_n\}$ est l'ensemble des poids des applications. Dans notre contexte, p_i représente le temps écoulé depuis le début de l'exécution de l'application i. p_i est en fait le temps de calcul non encore sauvegardé de l'application i.
$S = \{s_1, ..., s_n\}$, où s_i représente le volume mémoire occupé par l'application i.
$\delta = \{\delta_1, ..., \delta_n\}$, où $\delta_i = 1$ si l'application i est sélectionnée et $\delta_i = 0$ sinon.
n_δ est le nombre d'applications sélectionnées c'est-à-dire telles que $\delta_i = 1$; $n_\delta = \sum_{i=1}^{n} \delta_i$.
Nous utilisons la fonctions bw présentée au chapitre 4 pour estimer la bande passante $bw(m, V)$ nécessaire pour la sauvegarde en parallèle de m applications de volume mémoire agrégé V. Nous supposons pour la suite que pendant le processus de sauvegarde des applications, la bande passante globale $bw(m, V)$ est équitablement répartie entre les m tâches, c'est-à-dire qu'à chaque application est attribuée une bande passante $bw(m, V)/m$.

Avec ces notations, le problème de la sélection des applications à sauvegarder dans un délai T est formulé comme suit :

$$\begin{cases} \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad maximiser \ \sum_{i \in I} p_i \delta_i \\ sous\ contrainte\ (\max\{s_i\delta_i\})/[bw(n_\delta, \sum_{i \in I} s_i\delta_i)/n_\delta] \leq T \end{cases} \tag{5.3}$$

où
- $\max\{s_i\delta_i\}$ représente le plus grand volume mémoire occupé parmi les n_δ applications sélectionnées.
- $bw(n_\delta, \sum_{i \in I} s_i\delta_i)/n_\delta$ est la bande passante estimée, dédiée à la sauvegarde de chacun des n_δ applications sélectionnées.
- $(\max\{s_i\delta_i\})/[bw(n_\delta, \sum_{i \in I} s_i\delta_i)/n_\delta]$ représente le temps estimé pour la sauvegarde des n_δ applications.

Le problème ainsi formulé est complexe et nous en déduisons le théorème présenté dans la section qui suit.

5.3.2 Théorème

Théorème 1

Le problème (5.3) est NP-Complet.

Preuve : Il suffit de montrer que le problème (5.3) peut se réduire en un problème NP-Complet.

Dans (5.3), une première contrainte explicite pour la sauvegarde en parallèle de $n = |I|$ applications est la bande passante du système.
Supposons cette bande passante suffisamment grande pour pouvoir sauvegarder toutes les n

applications à la fois. Dans ce cas notons $bw\left(n, \sum_{i=1}^{n} s_i\right) = B$. Ainsi, une bande passante de B/n est assignée à chacune des n applications pour leur sauvegarde.
Notons également S_δ la somme des tailles des applications sélectionnées c'est-à-dire telles que $\delta_i = 1$; $S_\delta = \sum_{i=1}^{n} \delta_i s_i$.
B est la bande passante nécessaire pour la sauvegarde en parallèle des n applications sans perte de performance du système. D'après l'étude menée à la section 4.3.4, on a

$$\forall \delta \in \{0,1\}^n, bw(n_\delta, S_\delta) \leq B$$

ce qui implique que

$$\frac{1}{B} \leq \frac{1}{bw(n_\delta, S_\delta)}, \text{ par conséquent } \frac{\max\{s_i\delta_i\}}{B/n_\delta} \leq \frac{\max\{s_i\delta_i\}}{bw(n_\delta, S_\delta)/n_\delta}$$

En considérant ces inéquations, la contrainte du problème (5.3) devient alors

$$\frac{\max\{s_i\delta_i\}}{B/n_\delta} \leq T, \text{ ce qui implique que } \forall i \in I, \frac{s_i\delta_i}{B/n_\delta} \leq T \text{ i.e. } s_i\delta_i \leq \frac{TB}{n_\delta}$$

Puisque $n_\delta \leq n$, on a $\dfrac{TB}{n} \leq \dfrac{TB}{n_\delta}$
La contrainte $s_i\delta_i \leq \dfrac{TB}{n_\delta}$ est donc satisfaite si $s_i\delta_i \leq \dfrac{TB}{n}$ est satisfaite.
En sommant sur tous les éléments de I, on a :

$$\sum_{i=1}^{n} s_i\delta_i \leq \sum_{i=1}^{n} \frac{TB}{n} \text{ i.e. } \sum_{i=1}^{n} s_i\delta_i \leq TB$$

Puisque la bande passante est équitablement répartie, le problème (5.3) se ramène à :

$$\begin{cases} \quad maximiser \sum_{i \in I} p_i\delta_i \\ sous\ contrainte \sum_{i \in I} s_i\delta_i \leq TB \end{cases} \tag{5.4}$$

TB représente le volume de données que l'on peut sauvegarder pendant le temps T en utilisant une bande passante B.

Ainsi sous l'hypothèse que la bande passante du système est suffisamment grande pour assurer la sauvegarde des applications sans perte de performance du dispositif de sauvegarde, le problème (5.3) se réduit en un problème du sac-à-dos 0/1 classique qui est NP-complet. Le problème (5.3) est donc NP-complet. ■

De la preuve de la NP-Complétude du problème de la sélection des applications à sauvegarder dans un délai T, nous constatons que ce problème est plus complexe que celui du sac-à-dos 0/1 classique où la contrainte est juste la somme des tailles des objets sélectionnés. Pour (5.3), nous proposons, dans la section qui suit, une approche basée sur l'algorithme (*SS-Glouton*) proposé par Sartaj Sahni dans [8], pour ce problème dont la contrainte est non linéaire.

5.3.3 Approche d'approximation

Dès qu'une collection $\{i_1, i_2, ..., i_r\}$ est sélectionnée en résolvant (5.3), la sauvegarde des applications est engagée. La tâche $\bar{i}$ qui minimise $(s_i\delta_i)/[bw(n_\delta, \sum_{i\in I} s_i\delta_i)/n_\delta]$ sera sauvegardée pendant le temps $\bar{t} = (s_{\bar{i}}\delta_{\bar{i}})/[bw(n_\delta, \sum_{i\in I} s_i\delta_i)/n_\delta]$ pendant que la sauvegarde des autres processus se poursuivra. Nous supposons ci-après que $s_{i_1} \geq s_{i_2} \geq, ..., \geq s_{i_r}$. Par conséquent $\bar{i} = i_r$. Afin d'utiliser la bande passante libérée par la tâche $\bar{i}$, celle-ci doit être remplacée par une tâche $i \notin \{i_1, i_2, ..., i_r\}$. Ceci est réalisé en résolvant une autre instance du problème (5.3) sur le nouvel ensemble des tailles en mémoire $I' = I - \{\bar{i}\}$ définit tel que

$$s_i^{'} = \begin{cases} s_i - s_{\bar{i}} & \text{si } i \in \{i_1, i_2, ..., i_r\} \\ s_i & \text{sinon} \end{cases}$$

Cependant, puisque les sauvegardes en cours ne sont pas interrompues, les tâches $i \in \{i_1, i_2, ..., i_{r-1}\}$ sont prioritaires.
Le diagramme du schéma d'approximation est donné à la figure 5.2.

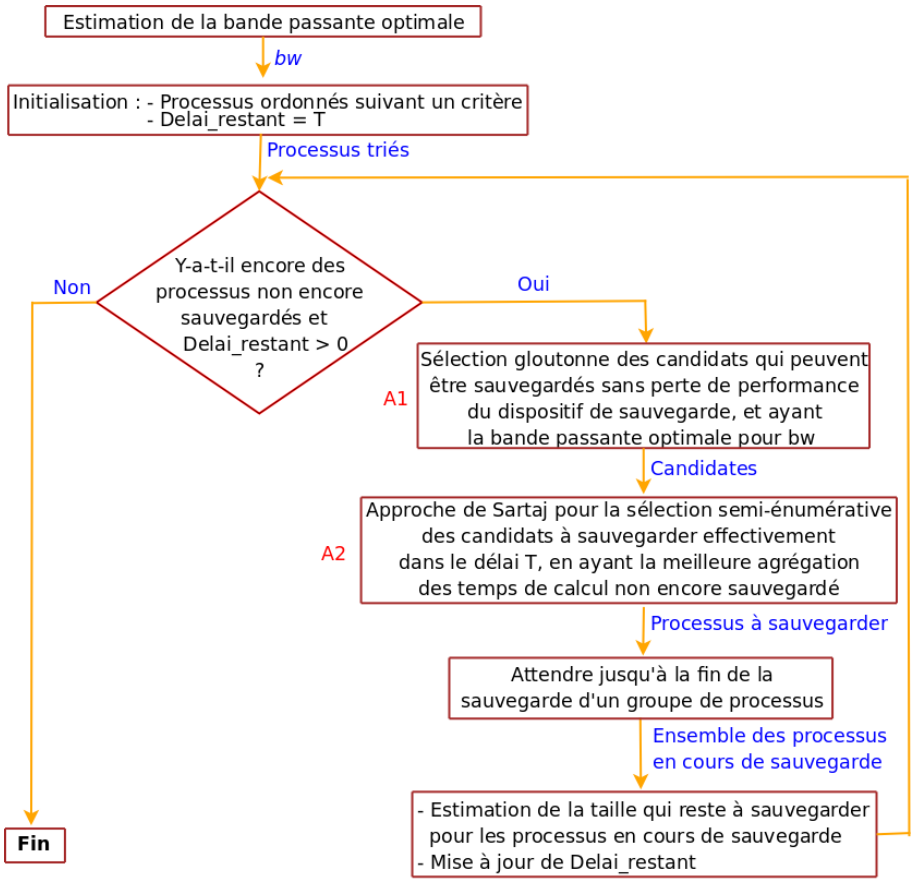

FIGURE 5.2 – Diagramme du schéma d'approximation

L'algorithme complet qui a été résumé dans le schéma d'approximation est donné dans la section qui suit.

5.4 Algorithme d'ordonnancement des sauvegardes

Dans cette section nous présentons en détail l'algorithme d'ordonnancement des sauvegarde et nous donnons une trace d'exécution de ses principales procédures.

5.4.1 Algorithme d'ordonnancement

A la section 5.3.1, nous avons proposé une formulation du problème d'ordonnancement des sauvegardes comme une séquence de variantes du problème du sac-à-dos 0/1. Cette séquence est implémentée sous forme de boucle qui s'exécute tant que le délai imposé T n'est pas écoulé et et qu'il y a encore des applications à checkpointer. Il est important de noter que la sauvegarde d'une application se fait sans interruption. Toutefois, la sauvegarde d'une application donnée peut être étalé sur plusieurs passages dans la boucle.

Pour réaliser cet objectif, nous procédons comme suit : (i) à chaque itération, déterminer, en utilisant une stratégie gloutonne, une collection I de n_0 processus, $n_0 \leq n$, qui peuvent être sauvegardés simultanément sans perte de performance du dispositif de sauvegarde ; (ii) ordonnancer suivant une approche gloutonne, les sauvegardes d'applications d'un sous-ensemble de I tout en prenant en compte la contrainte T.

Dans l'algorithme 1 dénommé *ScheduleCkpt* qui suit, k_0 est le paramètre introduit à la section 5.1.

Algorithme 1 : *ScheduleCkpt* //Ordonnancement des sauvegardes

Initialisation : $t_{ckpt} \leftarrow 0$; $CKPT \leftarrow \emptyset$; $E' \leftarrow E$
// S' *dénote l'ensemble des tailles des processus de* E'
$sort(E')$
repeat
  $I \leftarrow index\_bw\_max(E')$
  $\delta \leftarrow ks\_bw(I, E', S', CKPT, T - t_{ckpt}, k_0)$
  if $\neg is\_null(\delta)$ ***or*** $CKPT \neq \emptyset$ **then**
    $CKPT \leftarrow CKPT \cup \{i \in I \ / \ \delta_i = 1\}$
    Démarrer la sauvegarde de tous les processus nouvellement insérés dans $CKPT$
    Soit $K \leftarrow \{i \in CKPT$ tel que $end\_of\_ckpt(i) = true\}$
    $CKPT \leftarrow CKPT - K$
    $s\_max \leftarrow max_{k \in K}\{s'_k\}$
    for ***each*** $j \in CKPT$ **do**
      $s'_j \leftarrow s'_j - s\_max$
    end
    $E' \leftarrow E' - K$
    $mettre\_a\_jour(tckpt)$
  end
until $is\_null(\delta)$ ***and*** $CKPT = \emptyset$

Deux critères peuvent être adopté pour trier l'ensemble des applications : (a) dans l'ordre décroissant des temps de calcul non encore sauvegardés p_i ; (b) dans l'ordre décroissant des densités p_i/s_i.

L'algorithme commence par une phase d'initialisation. Le temps écoulé t_{ckpt} depuis le début de l'ordonnancement est initialisé à zéro, et l'ensemble $CKPT$ des processus en cours de sauvegarde est initialisé à $\emptyset$ car il n'y a encore aucune sauvegarde engagée. L'ensemble des applications candidates E' est initialisé à l'ensemble des applications E.

Considérons maintenant la boucle *repeat*. L'ensemble I des processus qui peuvent être sauvegardés tout en préservant les performances du système est déterminé à la ligne 5 suivant la procédure gloutonne suivante :

Algorithme 2 : $index\_bw\_max(E')$ //Sélection des candidats

```
1 //Les candidats i = 1, ..., m de E' sont supposés ordonnés ;
2 I' ← CKPT ; i ← 1 ;
3 s' ← taille agrégée des processus appartenant à I' ;
4 while (i ≤ m) and (bw(| I' |, Σ_{j∈I'} s_j) ≤ bw(| I' ∪ {i} |, Σ_{j∈I'∪{i}} s_j)) do
5 |   I' ← I' ∪ {i} ;
6 |   s' ← s' + s_i;
7 |   i ← i + 1;
8 end
9 return I' ;
```

Cependant, parmi les processus sélectionnés I, seuls ceux qui peuvent être sauvegardés avant la fin du délai $T - t_{ckpt}$, seront effectivement sauvegardés. La procédure ks_bw réalise cela suivant l'approche adoptée dans [8] avec le paramètre k_0, tout en donnant priorité aux processus de $CKPT$ qui sont en cours de sauvegarde.

La ligne 10 est exécutée dès que la sauvegarde d'un groupe de processus $i \in K$ est achevée. Les lignes 11 à 16 se chargent de l'estimation des blocs mémoires qui restent à sauvegarder pour les processus en cours de sauvegarde. Le reste est facile à comprendre.

Notons que la procédure $index\_bw\_max$ à la ligne 5 résoud le problème énoncé dans la boite **A1** de la figure 5.2 et la procédure ks_bw à la ligne 6 résoud le problème énoncé dans la boite **A2**.

Le point focal de la complexité de l'algorithme *ScheduleCkpt* est la procédure ks_bw. A chaque boucle i de l'algorithme *ScheduleCkpt*, notons $r_i =\mid I \mid$. Dans [8], il est montré que la complexité de la procédure ks_bw est bornée supérieurement par $O(k_0 r_i^{k_0+1})$. Si l'agorithme *ScheduleCkpt* s'exécute en m passages dans la boucle, alors en notant $r = \max_{1 \leq i \leq m} r_i$, la complexité de l'agorithme *ScheduleCkpt* est en temps polynomial borné supérieurement par $O(\sum_{i=1}^{m} k_0 r^{k_0+1}) = O(m k_0 r^{k_0+1})$.

L'algorithme détaillé ks_bw est donné en annexe A. Nous présentons dans la section qui suit, l'application de l'algorithme d'ordonnacement des sauvegardes sur un exemple. Nous présentons plus précisément l'exécution des deux principales procédures : $index\_bw\_max$ et ks_bw.

5.4.2 Exemple : trace d'exécution de l'algorithme d'ordonnancement

Nous avons implémenté l'algorithme d'ordonnancement que nous avons testé sur la grappe Azur de Sophia. L'exemple suivant donne une trace des résultats des procédures *index_bw_max* et *ks_bw* de l'algorithme d'ordonnancement exécuté avec $n = 8$ processus dont les tailles sont comprises entre 100 Mo et 500 Mo. Le délai fixé est $T = 60s$, et
$E = \{1, 2, 3, 4, 5, 6, 7, 8\}$
$P = \{2850, 2650, 2550, 2480, 2370, 2280, 2250, 2180\}$
$S = \{253, 318, 140, 215, 270, 241, 127, 491\}$

Pour des raisons de simplicité, les ensembles E, P et S sont ordonnés suivant les valeurs décroissantes des temps de calcul non encore sauvegardé (valeurs de P). L'appel de la procédure *index_bw_max* génère dans I les indices de processus qui peuvent être sauvegardés sans perte de performance du système global en ayant une bande passante maximale. Ceci est présenté dans le tableau 5.1.
L'ensemble des candidats à la sauvegarde déterminé par *index_bw_max* est $I = \{1, 2, 3, 4, 5, 6, 7\}$.

indices Sel.	**np**	**Taille agrégée (Mo)** S_{np}	$bw(np, S_{np})$ (Mo/s)
1	1	253	6.00
1,2	2	571	13.11
1,2,3	3	711	17.24
1,2,3,4	4	926	20.94
1,2,3,4,5	5	1196	24.10
1,2,3,4,5,6	6	1437	26.67
1,2,3,4,5,6,7	7	1564	28.78

TABLE 5.1 – Sélection des candidats pour le checkpointing

Le processus 8 n'est pas sélectionné car $bw(8, 1564+491) = 28,52$ qui est inférieur à $bw(7, 1564)$, indiquant ainsi des pertes de performance.

Pour le cas particulier de ce test, nous constatons que $bw(8, 2055) = 28,52$ est bien supérieure aux valeurs de la bande passante avant le pic, et l'on serait tenter de se demander pourquoi ne pas aller plus loin. Il est alors important de noter qu'après le pic il est difficile de prédire la réaction du système global face à ce signe de dysfonctionnement, d'où la prudence de ne s'intéresser qu'aux valeurs de la bande passante qui n'indiquent pas des pertes de performance.

La procédure *ks_bw* est ensuite appelée avec l'ordre k_0, et la liste des processus qui doivent être effectivement sauvegardé est enregistrée dans δ. Les résultats suivants ont été obtenus avec $k_0 = 1$ et $k_0 = 2$ respectivement.
Pour $k_0 = 1$, $\delta = (1, 1, 1, 1, 1, 0, 0)$ et le temps de calcul non encore sauvegardé agrégé est $PMAX = 12900$. Pour $k_0 = 2$, $\delta = (1, 0, 1, 1, 1, 1, 1)$ et $PMAX = 14780$.

Les tableaux 5.2 et 5.3 présentent une trace d'exécution de la procédure *ks_bw*.

Num	Taille (Mo)	État (Sauvegardé=1)	Temps du ckpt (s)
1	253	1	49
2	318	1	58
3	140	1	29
4	215	1	41
5	270	1	54
6	241	0	0
7	127	0	0
8	491	0	0

TABLE 5.2 – Trace du résultat avec $k_0 = 1$

Num	Taille (Mo)	Etat (Sauvegardé=1)	Temps du ckpt (s)
1	253	1	54
2	318	0	0
3	140	1	31
4	215	1	48
5	270	1	59
6	241	1	50
7	127	1	28
8	491	0	0

TABLE 5.3 – Trace du résultat avec $k_0 = 2$

5.5 Conclusion

Dans ce chapitre nous avons présenté une formulation du problème de la sauvegarde en parallèle d'applications, avec des contraintes de bandes passante et une contrainte temporelle, dans une grappe virtuelle tout en exploitant efficacement la grappe, comme une variante du problème du sac-à-dos 0/1 où la contrainte intègre la fonction *bw* présentée au chapitre 4. Nous avons démontré que le problème d'ordonnancement ainsi formulé est NP-Complet, ce qui a permis d'envisager une solution approchée du problème.

Nous avons proposé un algorithme d'ordonnancement qui produit des solutions approchées du problème posé et qui régule les sauvegardes d'applications. Cet algorithme fonctionne sous forme de boucle, en deux grandes phases à chaque boucle. A chaque passage, l'algorithme d'ordonnancement sélectionne de façon gloutonne dans un premier temps les candidats à sauvegarder sans perte de performance du dispositif de sauvegarde, puis dans un second temps une approche semi-énumérative est utilisée pour déterminer les applications à sauvegarder dans le délai imposé, puis suivent les sauvegardes proprement dites.

Le modèle présenté dans ce chapitre s'avère bien indiqué pour une exploitation efficace des ressources d'un intranet comme grappe virtuelle de calcul à de longues périodes d'inactivité des postes de travail. La préemption assurée par les mécanismes de sauvegarde/reprise permet ainsi de reprendre les applications inachevées sur éventuellement d'autres postes de travail lorsque les ressources deviennent à nouveau disponibles. Il est clair que pour éviter toute perturbation

du fonctionnement du modèle proposé, la grappe virtuelle devra être dédiée aux applications considérées.

Pour apprécier la validité de ce modèle, nous présentons dans la partie qui suit, l'implémentation et l'intégration du modèle d'ordonnancement dans un gestionnaire de ressources et nous évaluons ses performances en émulant un intranet managé par le gestionnaire de ressource.

Troisième partie

Intégration dans un gestionnaire de ressources

Chapitre 6

Principaux éléments d'implémentation

Sommaire

La préemption est très importante et largement utilisée dans les gestionnaires de ressources. Les systèmes comme Condor et SGE utilisent les mécanismes du checkpointing pour sauvegarder le contexte d'exécution d'une application qui s'exécute sur une machine volontaire, pour libérer la ressource au retour de l'utilisateur principal. L'application est alors sauvegardée sur un serveur dédié puis redémarrée sur une autre machine disponible. Comme nous l'avons vu au chapitre 2 , cette approche de préemption est plus efficace pour l'exploitation des périodes de micro-inactivité.

Nous avons proposé un modèle de sauvegarde en parallèle des applications de calcul haute performance qui s'exécutent dans une grappe virtuelle construite à partir des ressources libres pour de longues périodes d'un intranet. Ce modèle de préemption revêt encore plus d'importance s'il peut être appliqué dans la réalité. C'est ce que nous présentons dans ce chapitre.

Le présent chapitre est dédié à l'intégration du modèle d'ordonnancement proposé au chapitre 5 dans un gestionnaire de ressources, en l'occurrence *OAR*. Le prototype conçu est implémenté de façon à interagir avec la base de données de OAR, indépendamment des modules de OAR. De ce fait, ce prototype est portable et peut être aisément modifié pour interagir avec tout autre gestionnaire de ressource que OAR, du moment que la structure de la base de données est connue et qu'il existe une fonction d'estimation de la bande passante nécessaire pour la sauvegarde en parallèle d'un groupe de processus dans l'environnement considéré.

Nous commençons tout d'abord par présenter l'architecture du gestionnaire des ressources et de travaux OAR. Cette présentation permettra de mieux comprendre comment le prototype implémenté de l'ordonnanceur des sauvegardes interagit avec la base de données OAR.

6.1 Architecture du gestionnaire de ressources

Le gestionnaire de ressources dont il est question ici est OAR, qui est développé au laboratoire LIG sous forme de paquet linux sous licence GNU GPL. Son architecture est basée sur 3 principaux modules : *oar_server*, *oar_user* et *oar_node*.

oar_server est le paquet de OAR chargé de la gestion des ressources et des tâches administratives. Installé sur le serveur de la grappe, il permet de faire des mises à jour sur la base de données (ajout/suspension/suppression d'une ressource, validation des requêtes des utilisateurs en fonction de la disponibilité des ressources ...). Une liste exhaustive des activités de *oar_server* est fournie dans le *document technique*[1] de OAR.

oar_user est le paquet qui gère les requêtes des utilisateurs. Il est généralement installé sur la frontale qui est une machine de la grappe servant entre autre à authentifier les utilisateurs.

oar_node est le paquet installé sur les nœuds de calcul. Il collabore avec *oar_server* et *oar_user* pour l'exécution et la terminaison des tâches, ainsi que d'autres échanges intrinsèques au fonctionnement de OAR.

L'interconnexion des composantes de OAR présentant leur fonctionnement dans une grappe

1. http ://oar.imag.fr/admins/admin_documentation.html

de l'infrastructure Grid5000 est illustrée à la figure 6.1.

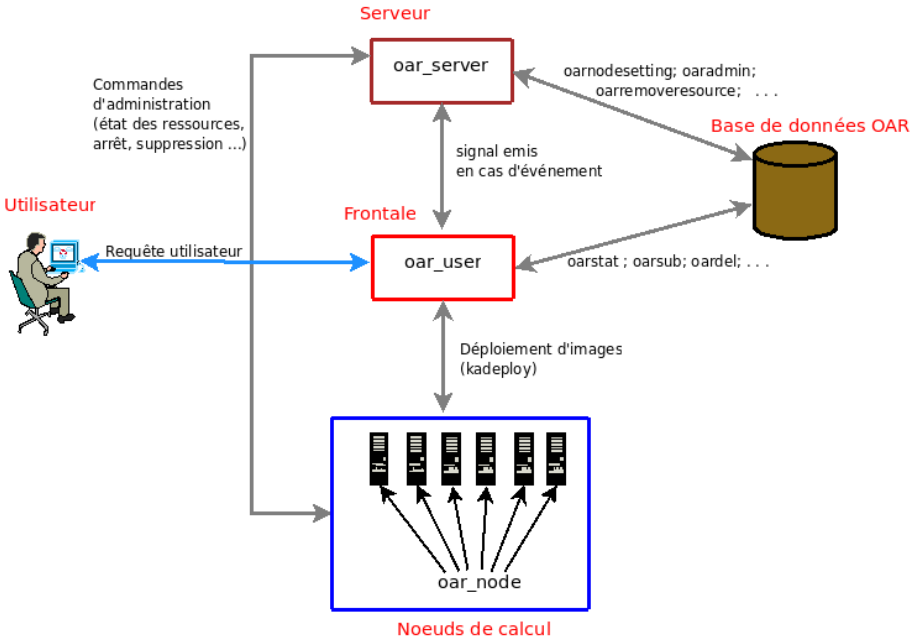

FIGURE 6.1 – Fonctionnement de OAR dans une grappe

Lorsqu'un utilisateur émet une requête, *oar_user* enregistre celle-ci dans la base de données de OAR et informe *oar_server*. Le gestionnaire de ressources *oar_server* consulte la base de données et vérifie s'il y a assez de ressources pour satisfaire à la requête. Dans le cas favorable ou non, la requête est mise à jour et *oar_server* émet un signal qui informe *oar_user* de la fin du traitement de la requête. *oar_user* consulte la base de données et informe l'utilisateur sur l'état de sa requête. Dans le cas favorable, la requête de l'utilisateur est exécutée au temps indiqué sur les nœuds de calcul alloués, où des images sont déployées. Les images peuvent être celles qui existent par défaut dans l'environnement de Grid5000 ou bien celles créées par l'utilisateur lui-même à travers l'outil *Kadeploy*[2].

Avec cette présentation des composantes de OAR et de leur fonctionnement dans une grappe, nous pouvons maintenant nous intéresser à l'architecture de l'ordonnanceur des sauvegardes que nous proposons.

6.2 Implémentation de l'ordonnanceur des sauvegardes

Le prototype de l'ordonnanceur des sauvegardes est implémenté en C sous forme d'une application client/serveur multithread basée sur le modèle *un thread par client accepté* [10],

2. https ://gforge.inria.fr/projects/kadeploy3/

dont l'architecture est présentée à la figure 6.2. Il correspond à la description du problème formulé à la section 5.2 c'est-à- dire une application par nœud.

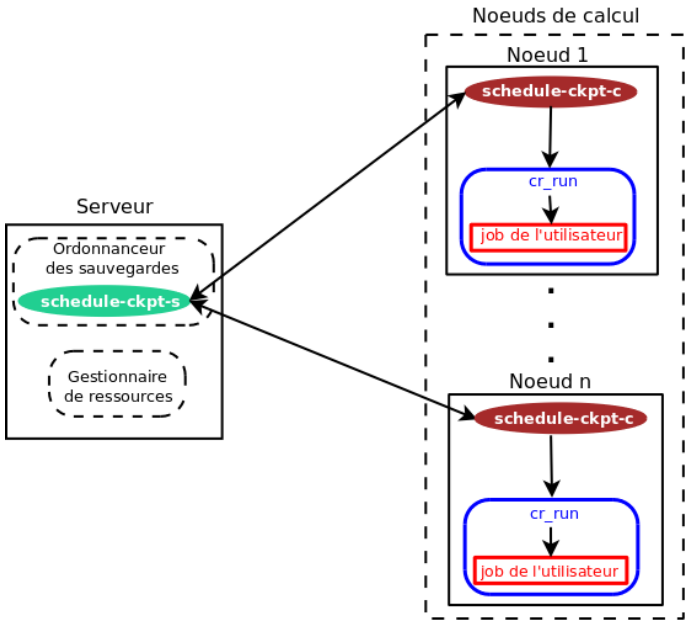

FIGURE 6.2 – Architecture de l'ordonnanceur des sauvegardes

Le choix d'une telle architecture a été motivé par plusieurs raisons :

- certaines informations nécessaires pour l'ordonnancement des sauvegardes, comme l'espace mémoire occupé par une application, ne figurent pas dans la base de données de OAR. L'application cliente de l'ordonnanceur des sauvegardes est donc chargée de récupérer ces informations et de les transmettre à l'application serveur de l'ordonnanceur des sauvegardes.
- pour qu'une application soit sauvegardée avec BLCR, il faudrait qu'elle soit préparée à la sauvegarde avec certains modules de BLCR. L'application cliente de l'ordonnanceur des sauvegardes est chargée de cette préparation. Ceci permet d'affranchir l'utilisateur des aspects techniques.
- cette architecture permet de tirer avantage de la puissance de calcul fournie par les nœuds qui sont multiprocesseurs multicœurs. En effet l'application cliente de l'ordonnanceur des sauvegardes permettra d'exécuter plusieurs applications à la fois sur un même nœud, comme on le verra à la section 6.4 où nous présentons une extension de l'architecture proposée.

Il convient de présenter les deux composantes de l'ordonnanceur des sauvegardes, que sont *schedule-ckpt-c* et *schedule-ckpt-s*.

6.2.1 Module de soumission des tâches préparées à la sauvegarde

L'application cliente de l'ordonnanceur des sauvegardes *schedule-ckpt-c* est exécutée sur les nœuds de calcul par le gestionnaire des travaux de OAR comme tâche de l'utilisateur. En fait lors de la soumission d'une requête, si l'utilisateur souhaite que son application soit prise en compte pour une éventuelle sauvegarde, il devra la passer comme paramètre à *schedule-ckpt-c*. Lorsque *schedule-ckpt-c* est exécuté, il crée un processus fils dont le contexte d'exécution est remplacé par celui de l'application qu'il exécute (celle de l'utilisateur) en la préparant à la sauvegarde avec BLCR. Ceci permet à l'application cliente de l'ordonnanceur des sauvegardes d'obtenir le *PID* du processus à sauvegarder. *schedule-ckpt-c* se sert alors de ce *PID* pour obtenir la taille en mémoire de l'application et d'autres informations nécessaires à l'ordonnancement.

6.2.2 Module d'ordonnancement sur le serveur

L'application serveur de l'ordonnanceur des sauvegardes *schedule-ckpt-s* est installée sur le serveur de la grappe et cohabite avec le gestionnaire de ressource *oar_server* de OAR. Il est chargé de lire les informations concernant les tâches en cours d'exécution, dans la base de données de OAR. Ces informations sont complétées avec celles reçues de l'application cliente de l'ordonnanceur des sauvegardes pour assurer l'ordonnancement des sauvegardes suivant le modèle proposé au chapitre 5. Nous présentons plus en détail les informations manipulées ainsi que le fonctionnement multithreadé de *schedule-ckpt-s* à la section 6.3.

Dès lors que les composantes du gestionnaire de ressources OAR et de l'ordonnanceur des sauvegardes ont été présentées, nous montrons dans la section qui suit, comment le tout opère au sein d'une grappe virtuelle.

6.3 Intégration de l'ordonnanceur des sauvegardes dans le gestionnaire de ressources

L'ordonnanceur des sauvegardes n'est pas intégré dans OAR sous forme d'un module, mais sous forme d'une application indépendante qui dépend des informations provenant de OAR. Ce choix est fait pour des raisons de portabilité du prototype, comme nous l'avons dit au début de ce chapitre.

Dans cette section, nous présentons le schéma d'intégration de l'ordonnanceur des sauvegardes dans une grappe virtuelle managée par OAR et nous expliquons le fonctionnement de l'ensemble de l'architecture.

6.3.1 Fonctionnement du gestionnaire de ressources avec sauvegarde

Dans les environnements de type grappe, où les machines sont dédiées pour le calcul scientifique, les serveurs sont très sollicités. Pour minimiser le taux de pannes dans le système,

certains administrateurs adoptent d'installer les différents serveurs de services sur des machines virtuelles du serveur de la grappe. Par exemple la figure B.1 de l'annexe B montre une telle politique sur Grid5000. *oar_server* (1), *oar_user* (2), la base de données de OAR (3) et d'autres services sont sur des machines virtuelles XEN [50] différentes.

La politique de placement présentée dans le paragraphe précédent est intéressante pour des environnements dédiés, mais n'est pas d'une grande importance pour les environnements dynamiques comme les grappes virtuelles qui ne sont construites que temporairement. Par conséquent, pour simplifier et faciliter l'utilisation des ressources de l'intranet lorsqu'elles sont libres, nous proposons que *oar_server*, *oar_user* et la base de données de OAR soient tous installées directement sur le serveur de l'intranet où sont également copiés les exécutables de l'ordonnanceur des sauvegardes.

Dans le modèle d'intégration que nous proposons, lorsque la grappe virtuelle est créée, un certain nombre d'actions sont déclenchées suivant un ordre chronologique comme le montre la figure 6.3.

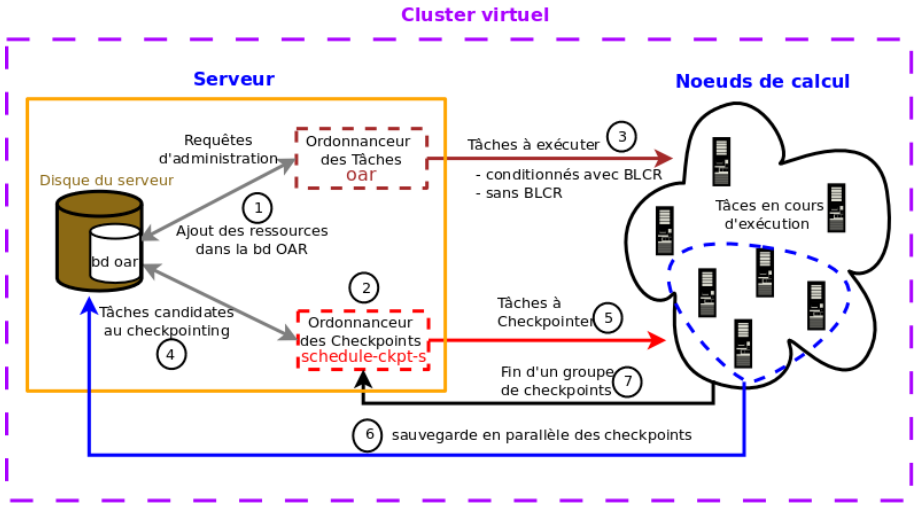

FIGURE 6.3 – Grappe virtuelle avec OAR et l'ordonnanceur des sauvegardes

Premièrement le gestionnaire de ressources *oar_server* est démarré, puis il enregistre les ressources dans la base de données de OAR à partir de la liste des machines disponibles de l'intranet dont il dispose.

Dans un deuxième temps, l'application serveur *schedule-ckpt-s* de l'ordonnanceur des sauvegardes est exécutée en prenant en paramètre la date de fin d'utilisation des ressources et le délai T imposé pour les sauvegardes. Après son exécution, *schedule-ckpt-s* s'endort pour se réveiller T secondes avant le début de l'ordonnancement des sauvegardes comme le montre la figure 6.4.

Ceci a pour avantage de ne prendre en compte que les applications en cours d'exécution, au moment où l'ordonnancement est initié. En effet, l'exécution de certaines applications peut s'achever avant que l'ordonnancement ne commence. Ces applications ne présentent donc aucun

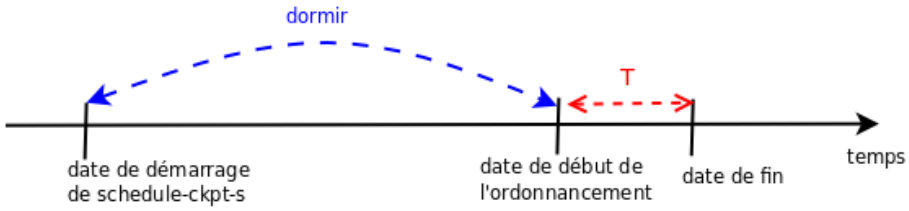

FIGURE 6.4 – Sommeil de *schedule-ckpt-s* avant le début de l'ordonnancement des sauvegardes

intérêt et il serait inutile de les pendre en compte dès le début de leur exécution. Il est important de souligner que les modules de BLCR doivent être chargés dans le noyau du système sur chaque nœud de calcul, suivant les indications de la documentation de la version de BLCR utilisée. Après ces deux premières phases, la grappe virtuelle dispose alors d'un environnement près à l'exécution des applications avec des possibilités de sauvegardes.

Avant de voir comment les tâches sont exécutées, il convient d'observer comment elles sont soumises au gestionnaire de travaux.

Soumission des tâches

L'ordonnanceur OAR permet d'effectuer des soumissions en deux modes :
- le *mode interactif*. Ici, l'utilisateur est connecté sur un nœud principal de la réservation lorsque des ressources lui sont allouées. Les résultats des exécutions sont directement affichés sur le terminal de l'utilisateur.
- le *mode passif*. Dans ce mode, les ressources sont réservées, mais l'utilisateur n'est pas directement connecté sur un nœud de la réservation. Un script spécifié lors de la réservation est exécuté. Les sorties standard et erreur du script sont redirigées vers les fichiers *OAR.JOB_ID.stdout* et *OAR.JOB_ID.stderr* du répertoire à partir duquel le script a été lancé.

Nous considérons pour la suite que les réservations se font en mode passif, car cela correspond bien à la soumission des travaux par lots et cela cadre également avec la configuration de la grappe virtuelle.

Le schéma général d'une réservation passive est le suivant :

oarsub -l /nodes=nombre_de_nœuds_souhaité,walltime=hh :mm :ss {script_utilisateur}

Le lecteur peut consulter la page Web de OAR2 [3] pour plus de détails sur les différentes formes de soumission.

Pour la soumission des tâches à OAR, nous distinguons deux cas : la soumission directe qui suit le schéma général précédemment présenté et la soumission via l'application cliente de l'ordonnanceur des sauvegardes *schedule-ckpt-s* dont le schéma est le suivant :

3. https ://www.grid5000.fr/mediawiki/index.php/OAR2_use_cases#JOB_submission

oarsub -l /nodes=nombre_de_nœuds_souhaité,walltime=hh :mm :ss {schedule-ckpt-c option <chemin>/app_utilisateur liste_des_arguments}

L'application de l'utilisateur est passée à *schedule-ckpt-s* suivant deux options : $-c$ qui permet de checkpointer une application qui n'a pas encore été sauvegardée, et $-r$ pour reprendre l'exécution d'une application à partir de son point de reprise. En effet ces exécutions sont bien distinctes car BLCR sauvegarde le point de reprise d'une application de PID *pid_app* dans un fichier nommé *context.pid_app* qui contient tout ce dont l'application a besoin pour la reprise, laquelle est effectuée par la commande *cr_restart*.

Exemples de soumission via *schedule-ckpt-c*
oarsub -l /nodes=1,walltime=06 :00 :00 {schedule-ckpt-c -c ./exemple-exe 100 500}
oarsub -l /nodes=1,walltime=11 :00 :00 {schedule-ckpt-c -r context.2134}

Après cette présentation de comment sont soumis les travaux, nous pouvons nous intéresser à la façon dont ils sont exécutés.

Exécution des travaux

Après les deux premières phases précédemment présentées, la phase d'exécution des travaux soumis est alors engagée. Les applications des travaux soumis, qui ont été passées comme arguments à *schedule-ckpt-c*, sont exécutées suivant le schéma présenté à la figure 6.5, après que *schedule-ckpt-c* soit lancé par OAR (via *oar_exec*).

Pour le cas d'une application candidate à la sauvegarde pour la première fois, *schedule-ckpt-c* crée un processus fils dont le contexte d'exécution est remplacé par celui de *cr_run* qui est exécuté avec comme argument l'application de l'utilisateur *App_utilisateur* et la liste éventuelle de ses arguments. La macro *cr_run* de BLCR exécute à son tour *App_utilisateur* en faisant un appel system *exec()* qui remplace son contexte par celui de *App_utilisateur*. A l'issue de cette succession de remplacements de contextes, seul *App_utilisateur* est en cours d'exécution, avec pour PID celui du processus fils de *schedule-ckpt-c*. Ceci permet ainsi à *schedule-ckpt-c* d'obtenir le PID de *App_utilisateur*.

Dans le cas de la reprise d'une application, la commande *cr_restart* de BLCR restaure le contexte d'exécution à partir du point de reprise *contexte.pid_initial* et le processus restauré a pour PID *pid_initial*. Ainsi, dans le cas de la reprise d'une application avec *schedule-ckpt-c*, le procédé décrit dans le paragraphe précédent est effectué jusqu'au remplacement du contexte de *cr_run* par celui de *cr_restart*. A partir de cette étape il n'y a plus de remplacement de contexte, car le PID de *cr_restart* est très fort probablement différent de *pid_initial*. Ainsi pour avoir le PID du processus redémarré, *schedule-ckpt-c* extrait *pid_initial* de *contexte.pid_initial*. Il est important de noter que la reprise d'une application est assimilable à l'exécution d'une nouvelle application, d'où la nécessité de la préparer de nouveau à la sauvegarde avec BLCR, car elle peut à nouveau être candidate. Cela est très intéressant pour des applications qui nécessitent de très long temps de calcul.

Dans les deux cas *schedule-ckpt-c* détient le PID de l'application de l'utilisateur. Il utilise

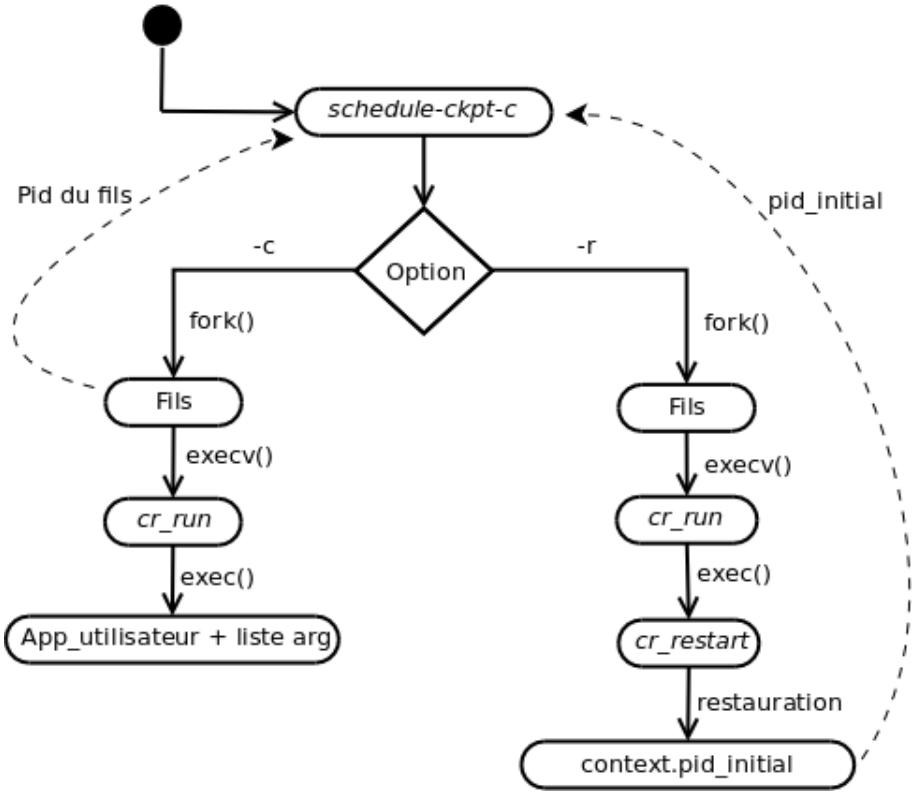

FIGURE 6.5 – Exécution des application avec *schedule-ckpt-c*

alors les informations contenues dans le système de fichiers */proc* pour obtenir les données dont a besoin *schedule-ckpt-s* au moment des sauvegardes. Par exemple la taille en mémoire physique d'une application de PID *pid_app* peut s'obtenir à partir du fichier virtuel */proc/pid_app/status*.

Chaque application soumise s'exécute sous le contrôle de *schedule-ckpt-c* qui se met en attente de message venant de *schedule-ckpt-s*. Au réveil de *schedule-ckpt-s*, l'étape d'ordonnancement des sauvegardes est engagée. La section qui suit présente le fonctionnement de la phase de sauvegarde.

6.3.2 Implémentation de la sauvegarde des tâches

L'étape d'ordonnancement des sauvegarde se déroule en trois phases : la collecte des données par *schedule-ckpt-s*, la phase de sauvegarde proprement dite et la phase de terminaison des soumissions.

Collecte des données

A son réveil, *schedule-ckpt-s* consulte la base de données de OAR dont le schéma est présenté à la figure C.1 de l'annexe C, pour extraire une partie des informations nécessaire à

l'ordonnancement.

La base de données de OAR est créée avec le système de gestion de bases de données MySQL. *schedule-ckpt-s* adresse des requêtes à la base de données de OAR au moyen des fonctions de la bibliothèque *libmysqld-dev* qui est est une API (Application Programming Interface) permettant d'accéder aux bases de données MySQL à travers du code en **C**.

Les tables de la base de données de OAR qui sont exploitées par *schedule-ckpt-s* sont : jobs, moldable_job_description, assigned_resources, resources. Les données principales extraites de ces tables sont :

- *job_id* : l'identifiant de la tâche
- *initial_request* : la ligne de commande complète de la requête de l'utilisateur
- *start_time* : la date de début de l'exécution de la tâche.
- *network_address* : nom du nœud qui va exécuter la tâche

Avec ces données *schedule-ckpt-s* détermine le temps de calcul non encore sauvegardé (p_i) de chaque tâche. Ensuite il reçoit les informations complémentaires venant de *schedule-ckpt-c* sur chaque nœud de calcul, notamment la taille en mémoire de l'application et le nom du nœud, pour faire les correspondances d'information.

Il est important de noter que cette phase s'exécute en un temps négligeable par rapport au délai T imposé. Si ce n'était pas le cas, on pourrait toujours s'arranger à ce que *schedule-ckpt-s* se réveille plus tôt.

Les données pour l'ordonnancement étant regroupées, la phase de sauvegarde est alors amorcée.

Phase de sauvegarde

La phase de sauvegarde se déroule suivant l'algorithme d'ordonnancement *ScheduleCkpt* proposé à la section 5.4. Le thread principal de *schedule-ckpt-s* détermine les premières applications à sauvegarder, cela correspond à l'exécution des lignes 5 et 6 de l'algorithme *ScheduleCkpt*. Ensuite il lance des threads pour assurer ces sauvegardes en leur passant les sockets des nœuds clients, à raison de un thread par application (algorithme *ScheduleCkpt* lignes 7 à 9), puis s'endort en attendant un signal de fin d'activité d'un groupe de threads.

Chaque thread contacte *schedule-ckpt-c* sur le nœud dont il a l'adresse et lui indique d'effectuer un checkpoint sur l'application dont il détient le PID, puis se met en attente d'une fin d'opération. *schedule-ckpt-c* effectue un appel system en invoquant la commande *cr_checkpoint* de BLCR pour assurer la sauvegarde, avec l'option de terminaison de l'exécution de l'application. A la fin de l'exécution de l'appel système, *schedule-ckpt-c* informe le thread qui émet un signal pour signifier au thread principal de la fin de sa mission.

Lorsque le thread principal est réveillé par un signal de fin d'activité (algorithme *ScheduleCkpt* ligne 10), il estime le volume de données qui reste à sauvegarder pour chaque processus en cours de sauvegarde et met à jour le temps restant imposé pour les sauvegardes (algorithme *ScheduleCkpt* lignes 11 à 17). Le procédé des deux paragraphes précédents est repris suivant l'algorithme d'ordonnancement jusqu'à ce qu'il n'y ait plus d'application à sauvegarder ou que

le délai restant n'est plus suffisant pour une sauvegarde (algorithme *ScheduleCkpt* ligne 19). Dans ce cas la phase de terminaison des tâches est alors lancée.

Terminaison d'une soumission

Si l'utilisateur soumet sa requête directement à OAR, celui-ci se charge d'arrêter la tâche lorsque le temps qui lui est alloué tire à sa fin. Par contre lorsque l'utilisateur soumet sa requête via l'application cliente de l'ordonnanceur des sauvegardes *schedule-ckpt-c*, cette dernière se charge de la terminaison de l'application suivant les ordres reçus de *schedule-ckpt-s*.

A la fin du delai imposé pour les sauvegardes, les tâches qui n'ont pas pu être sauvegardées doivent être arrêtées. *schedule-ckpt-s* envoie alors un message à chaque application cliente de l'ordonnanceur pour mettre fin à l'exécution de l'application dont il a le contrôle.

6.3.3 Synthèse

L'architecture de l'ordonnanceur des sauvegardes que nous avons proposé pour la soumission des tâches et la gestion des sauvegardes respecte la modélisation que nous avons faite suite au problème formulé à la section 5.2 où nous avons considéré une application par nœud.

Cette implémentation de l'ordonnanceur de sauvegarde est intégrée au gestionnaire de ressources OAR comme module indépendant. Elle peut être exploitable sur les grappes qui présentent le même environnement matériel que celui dans lequel *bw* a été déterminé, étant entendu que le modèle de sauvegarde proposé est fort dépendant de la détermination d'une estimation de la bande passante utilisée.

Pour tirer avantage de la puissance de calcul offerte par les nœuds qui sont de plus en plus multiprocesseurs multicœurs, dans la section qui suit, nous proposons une extension de l'architecture de l'ordonnanceur des sauvegardes. Nous montrons les contraintes que présente la nouvelle proposition dans certains cas.

6.4 Extension de l'architecture de l'ordonnanceur des sauvegardes aux processeurs multicœurs

L'architecture proposée précédemment ne permet d'exécuter qu'une seule application par nœud. Pour une utilisation efficace de la puissance de calcul dégagée par les nœuds de calcul dotés de plusieurs processeurs multicœurs, nous proposons un nouveau visage de l'architecture de l'ordonnanceur des sauvegardes, présenté à la figure 6.6.

Dans cette architecture, la structure de l'application cliente de l'ordonnanceur des sauvegardes *schedule-ckpt-c* a le plus visiblement changé. Les changements opérés sur *schedule-ckpt-s* sont effectués au niveau de l'implémentation pour qu'il s'accorde avec la nouvelle structure de *schedule-ckpt-c*.

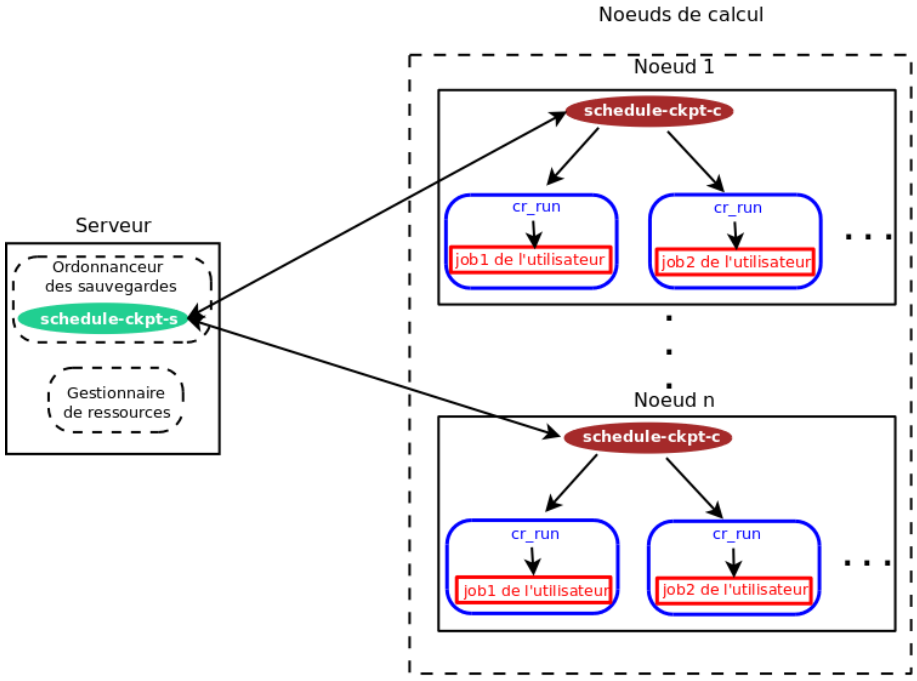

FIGURE 6.6 – Architecture étendue de l'ordonnanceur des sauvegardes

Avec cette nouvelle configuration, la soumission des tâches et l'exécution des sauvegardes sont élaborés avec quelques nouveautés.

6.4.1 Soumission des tâches

Avec cette nouvelle architecture, les soumissions de tâches via *schedule-ckpt-c* suivent maintenant cette syntaxe :

oarsub -l /nodes=nombre_de_nœuds_souhaité,walltime=hh :mm :ss {*schedule-ckpt-c* [option <chemin>/app_utilisateur liste_des_arguments]$^+$}

Nous adoptons la notation en langage formel $[expression]^+$ pour signifier que *expression* figure au moins une fois dans la formule. Cela signifie qu'à travers une requête plusieurs applications peuvent être soumises pour être toutes exécutées sur un même nœud, car chaque nœud sollicité exécutera le script passé entre accolades. Pour mieux comprendre cette syntaxe, nous donnons quelques exemples de soumission invoquant *schedule-ckpt-c*.

Exemples de soumission via *schedule-ckpt-c*

1. oarsub -l /nodes=1,walltime=05 :00 :00 {*schedule-ckpt-c* -c ./app1 fic_données1 }
2. oarsub -l /nodes=1,walltime=11 :00 :00 {*schedule-ckpt-c* -r context.9156}
3. oarsub -l /nodes=1,walltime=11 :00 :00 {*schedule-ckpt-c* -c app1 fic_données2 -r context.9156 }

4. oarsub -l /nodes=1,walltime=11 :00 :00 {*schedule-ckpt-c* -r context.9187 -c app2 400 600 -c app1 fic_données2 }
5. oarsub -l /nodes=1,walltime=11 :00 :00 {*schedule-ckpt-c* -c app3 -c app4 100 -c app2 400 600 -c app1 fic_données2 }

Lorsque *schedule-ckpt-c* est lancé, il exécute à son tour de façon séquentielle chacune des applications qui figurent dans sa liste d'arguments, de la même façon que décrit à la section 6.3.1. Par exemple si le cas (5) est exécuté sur un nœud biprocesseur bicœur, alors les quatre applications *app1*, *app2*, *app3* et *app4* seront exécutées sur les quatre cœurs du nœud, à raison d'une application par cœur. Ainsi, par rapport à l'architecture présentée à la section 6.2, on passe d'une application par nœud à quatre applications par nœud, ce qui montre une exploitation judicieuse de la puissance de calcul disponible. Pour une soumission dont l'identifiant est JOB_ID, les résultats des exécutions des applications issues de cette soumission sont tous enregistrés dans les mêmes fichiers $OAR.JOB\_ID.stdout$ et $OAR.JOB\_ID.stderr$.

Avec cette nouvelle architecture, le déroulement des sauvegardes est marqué par une activité plus intense de *schedule-ckpt-c* comme le montre la section suivante.

6.4.2 Déroulement de la sauvegarde avec la nouvelle architecture

Tout comme précédemment, le début de l'ordonnancement des sauvegardes est marqué par le réveil de *schedule-ckpt-s*. A son réveil, après avoir extrait certaines informations de la base de données de OAR comme précédemment, il reçoit maintenant de chaque application cliente *schedule-ckpt-c*, une structure de données qui contient les informations suivantes : la liste des applications en cours d'exécution sur le nœud concerné, le PID et la taille en mémoire de chacune de ces applications, le nom du nœud qui exécute ces applications. La phase de sauvegarde proprement dite peut alors être engagée. Cependant, avec cette nouvelle architecture, plusieurs démarches peuvent être adoptées au moment de la sauvegarde. Dans les sections qui suivent, nous en présentons deux que nous estimons fort intéressantes.

Sauvegarde en parallèle par vague

Une approche consiste à appliquer l'implémentation précédente de la sauvegarde décrite à la section 6.3.2 de façon répétitive sur chaque vague de processus provenant chacun d'un nœud différent, ceci pour conserver la configuration de la sauvegarde d'un processus par nœud. Les vagues de processus sont constituées comme le montre la figure 6.7. Dans cette figure, P_{ij} représente un processus en cours d'exécution sur le cœur j du nœud i.

Après avoir reçu les informations complémentaires de chaque application cliente *schedule-ckpt-c*, *schedule-ckpt-s* réorganise alors ses structures de données. Les processus de chaque nœud sont affectés dans des tableaux différents, chacun dans le tableau indicé par son rang, c'est-à-dire que chaque P_{ij} sera mis dans le tableau j. Chaque tableau correspond à une colonne de la figure 6.7. Ainsi la phase de sauvegarde décrite à la section 6.3.2 sera appliquée sur chaque tableau $j = 1, 2, ...$ tant que le délai imposé pour les sauvegardes n'est pas épuisé.

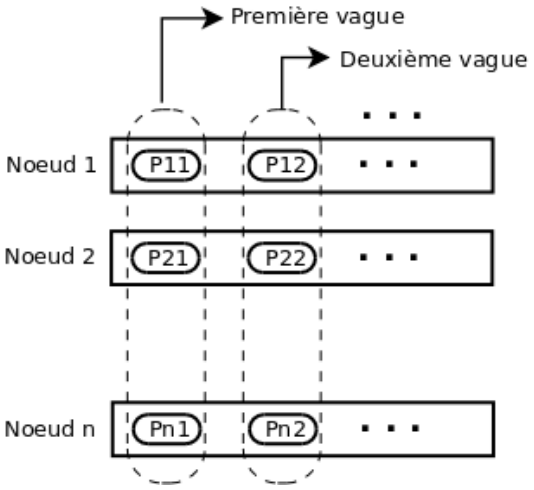

FIGURE 6.7 – Schéma de la sauvegarde en parallèle par vague

Cette approche permet d'exploiter la modélisation qui a été faite au chapitre 5, mais elle peut entraîner une sous utilisation de la bande passante. En effet, cette situation peut se présenter en fonction du nombre de processus dans une vague et de leur volume de données agrégé. Cependant elle permet d'utiliser au mieux la puissance de calcul disponible, de façon générale. Nous évaluerons cette implémentation dans le chapitre 7.

Une autre approche serait d'exploiter le parallélisme au sein d'un nœud pour assurer une sauvegarde en parallèle complète. Cette question est abordée dans la section qui suit.

Sauvegarde en parallèle complète

Dans cette approche, on considère que les processus d'un même nœud peuvent être sauvegardés simultanément.

Ici, après avoir reçu les informations complémentaires de *schedule-ckpt-c*, *schedule-ckpt-s* réorganise ses structures de données en affectant à chaque application les ressources qu'elle utilise. *schedule-ckpt-s* fonctionne alors comme s'il gérait la situation d'une application par nœud, sauf que maintenant un nœud peut recevoir plusieurs ordres de checkpoints simultanément. Avec cette nouvelle configuration *schedule-ckpt-c* est maintenant une application multithreads.

Comme à la section 6.3.2, *schedule-ckpt-s* détermine les premières applications à sauvegarder, puis lance un thread par application pour accomlir le service. Chaque application cliente de l'ordonnanceur des sauvegardes *schedule-ckpt-c* est en attente de message de *schedule-ckpt-s*. Lorsqu'un thread de *schedule-ckpt-s* contacte *schedule-ckpt-c*, il lui passe le PID du processus à sauvegarder. *schedule-ckpt-c* génère un thread à qui il confie la sauvegarde de l'application indiquée, puis il se remet en attente d'une éventuelle autre sollicitation de *schedule-ckpt-s*. Cela permet ainsi d'assurer la sauvegarde en parallèle au niveau de chaque nœud. Lorsque la sauvegarde d'une application est achevée, le thread de *schedule-ckpt-c* en charge envoie un message au thread de *schedule-ckpt-s* pour lui signifier la fin de sa mission. Ainsi le thread de *schedule-ckpt-s*

informe à son tour son thread principal de la fin de la sauvegarde.

Lorsque le thread principal de *schedule-ckpt-s* est réveillé par un signal de fin d'activité, le processus du paragraphe précédent est repris suivant l'agorithme d'ordonnancement jusqu'à ce qu'il n'y ait plus d'application à sauvegarder ou que le délai restant n'est plus suffisant pour une sauvegarde.

Cependant, pour évaluer la performance de la sauvegarde en parallèle au sein d'un nœud vers le disque du serveur, nous avons effectué un test en comparant les temps de sauvegarde d'une application séquentielle de volume à sauvegarder V au temps de sauvegarde en parallèle de $m \in \{2, 3, 4\}$ applications séquentielles s'exécutant sur le nœud, ayant chacune un volume à sauvegarder de V/m. Pour ce test nous avons utilisé BLCR-0.8.2 que nous avons installé dans une image créée, sous le noyau 2.6.30.2 de la distribution Debian de Linux. Notons que des améliorations ont été apportées dans les récentes versions de BLCR, ce qui permet d'optimiser les mécanismes de sauvegarde avec BLCR. Par conséquent les temps de sauvegarde réalisés avec BLCR-0.8.2 sont bien inférieurs à ceux réalisés avec BLCR-0.4.2 pour les mêmes applications sauvegardées. Les tests ont été effectués sur la grappe GDX de Orsay, sur des nœuds biprocesseurs bicœurs. Les résultats des tests sont présentés dans le tableau 6.1.

Nb. App	**Taille agrégée (Mo)**	**Temps de sauv. (s)**
1	300	11
3	3x100	14
1	400	12
2	2x200	16
1	600	17
3	3x200	21

TABLE 6.1 – Comparaison des temps de sauvegardes

Des données expérimentales présentées au tableau 6.1, il ressort que le temps de sauvegarde en parallèle de m applications de volume agrégé V (et donc de taille chacune V/m) est supérieur au temps de sauvegarde d'une seule application de même volume à sauvegarder V. Cela traduit un conflit d'utilisation du chemin de sauvegarde à l'intérieur du nœud, comme le montre la figure 6.8, lors d'une sauvegarde en parallèle.

Dans cette configuration, lors de la sauvegarde en parallèle des applications qui s'exécutent sur un même nœud, celles-ci doivent se partager la bande passante du bus mémoire et du bus PCI, ainsi que le buffer de la carte réseau. De plus, la concurrence pour l'utilisation des composantes du chemin de sauvegarde avec les processus des autres nœuds rend le problème encore plus complexe.

Un problème similaire a été présentée dans [96] où l'étude des communications concurrentes au cours de l'exécution d'applications MPI a révélé des pénalités, induites par la concurrence, sur les temps de communication. Le cas des conflits sortant/sortant(S/S) présenté dans [96] s'apparente au conflit induit par la sauvegarde en parallèle de deux applications. Cependant la différence réside au niveau des destinations. Dans [96] les conflits sont étudiés par rapport aux communications entre processus d'une application parallèle et le conflit S/S présenté est

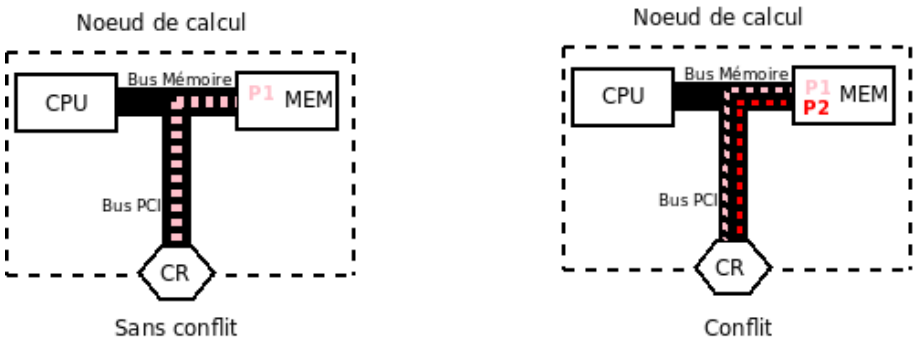

FIGURE 6.8 – Sauvegarde séquentielle/parallèle au sein d'un nœud

celui de l'envoi simultané d'un même message à deux processus distincts situés sur des nœuds différents. Dans le cas de notre étude, le conflit est généré par la sauvegarde en parallèle de deux applications, depuis la mémoire d'un nœud vers le disque d'un autre nœud.

Avec le conflit induit par la sauvegarde en parallèle de plusieurs applications s'exécutant sur un même nœud, l'utilisation de l'estimateur *bw* de la bande passante nécessaire pour la sauvegarde en parallèle sans perte de performance n'est plus indiquée. Il serait alors intéressant de mener une étude plus approfondie du nouveau problème posé en vue d'utiliser efficacement la nouvelle architecture proposée lors des sauvegardes.

6.5 Conclusion

Ce chapitre a eu pour objectif de montrer comment le modèle d'ordonnancement proposé dans cette thèse s'intègre dans un gestionnaire de ressources. Pour le cas d'espèce, l'intégration d'un prototype de l'ordonnanceur des sauvegardes dans le gestionnaire de ressources OAR des grappes de Grid5000 a été présenté.

Dans le souci d'exploiter efficacement la puissance de calcul dégagée par les nœuds qui sont désormais multiprocesseurs multicœurs, une extension de l'architecture de l'ordonnanceur des sauvegardes a été proposée. Cette nouvelle architecture permet d'exécuter plusieurs applications sur un même nœud en les préparant à la sauvegarde. Pour la phase de sauvegarde, deux approches on été présentées. La première permet de sauvegarder les applications en parallèle par vagues constituées de processus provenant de nœuds différents. Cette approche exploite directement l'implémentation initialement proposée. La seconde propose d'exploiter entièrement le parallélisme offert au sein d'un nœud pour la sauvegarde en parallèle d'applications. Cependant, une analyse de performance de cette approche a révélé des conflits intra-nœud lors de la sauvegarde en parallèle de plusieurs applications s'exécutant sur le nœud. Le problème de l'étude de ces conflits a été posé. Une piste serait d'expérimenter l'étude menée à la section 4.3.

L'intégration du modèle d'ordonnancement dans un gestionnaire de ressource ayant été présenté, il convient de voir comment ceci fonctionne et permet d'utiliser efficacement les ressources

libres d'un intranet pour faire du calcul haute performance. Cette question est abordée dans le chapitre suivant où nous évaluons l'implémentation de l'architecture étendue avec sauvegarde par vague.

Chapitre 7

Validation expérimentale

Sommaire

Ce chapitre a pour objectif de montrer les performances du modèle d'ordonnancement proposé, à travers une évaluation de l'implémentation intégrée au gestionnaire de ressources OAR, suivant le schéma d'intégration présenté faite dans le chapitre précédent.

Pour les tests, nous avons utilisé l'implémentation de l'architecture étendue de l'ordonnanceur des sauvegardes présentée à la section 6.4, avec la technique de sauvegarde en parallèle par vague indiquée au 6.4.2. Cela permet d'évaluer tant la sauvegarde avec une application par nœud qu'avec plusieurs applications par nœud.

Les résultats présentés ici ont été obtenus sur une plateforme réelle construite dans l'environnement GRID5000 pour réaliser les tests. Cette approche permet d'apprécier directement le comportement de l'implémentation de l'ordonnanceur des sauvegardes, afin de le porter aisément dans un intranet, à partir du moment où l'on dispose d'un estimateur de la bande passante nécessaire pour la sauvegarde sans perte de performance du dispositif de sauvegarde.

La section 7.1 présente la construction de la plateforme de test. Le scénario des expérimentations est décrit à la section 7.2. Ce chapitre s'achève à la section 7.3 avec la présentation des résultats et quelques discussions conséquentes.

7.1 Construction de la grappe de test

Nous avons eu la primeur d'utiliser la version 2.4.0-1 la plus récente de OAR, dont le développement venait de s'achever au moment où nous commencions l'implémentation du modèle d'ordonnancement des sauvegardes. La mise en ligne de OAR-2.4.0-1 était en cours, selon les informations fournies par les développeurs. Nous avons également utilisé l'une des versions les plus récentes de BLCR, BLCR-0.8.2 que nous avons installé dans l'image créée, sous la distribution Debian de Linux 2.6.30.2.

Pour construire un environnement de test réel qui soit proche des intranets, nous avons créé deux images avec l'outil *Kadeploy* à partir de l'image de base *sid-nfs-base* de l'environnement GRID5000. Ces images sont présentées à la figure 7.1.

Dans l'image *Image_oar_server* nous avons installé le module *oar_server* de OAR chargé de la gestion des ressources, le module oar_admin chargé des tâches de configuration, *oar_user* chargé de la gestion des requêtes des utilisateurs, ainsi que les paquets de dépendance et autres utilitaires. Les dossiers contenant les exécutables *schedule-ckpt-s* et *schedule-ckpt-c* de l'ordonnanceur des sauvegardes figurent également dans cette image. Elle est déployée sur le nœud qui servira en même temps de serveur OAR et de frontale. Dans un intranet, tous les paquets présentés dans l'image *Image_oar_server* devront être installés sur le serveur.

Dans l'image *Image_oar_node*, le module oar_node a été installé, ainsi que les paquets de dépendance. Cette image est déployée sur les nœuds de calcul.

Une fois les nœuds acquis après une réservation sur une grappe de GRID5000, les deux images présentées sont déployées, puis les étapes (1) et (2) de la figure 6.3 sont démarrées comme décrit à la section 6.3.1. Après le déploiement des images sur les différents nœuds, le serveur NFS est démarré sur le nœud faisant office de serveur de la grappe créée, puis son */home*

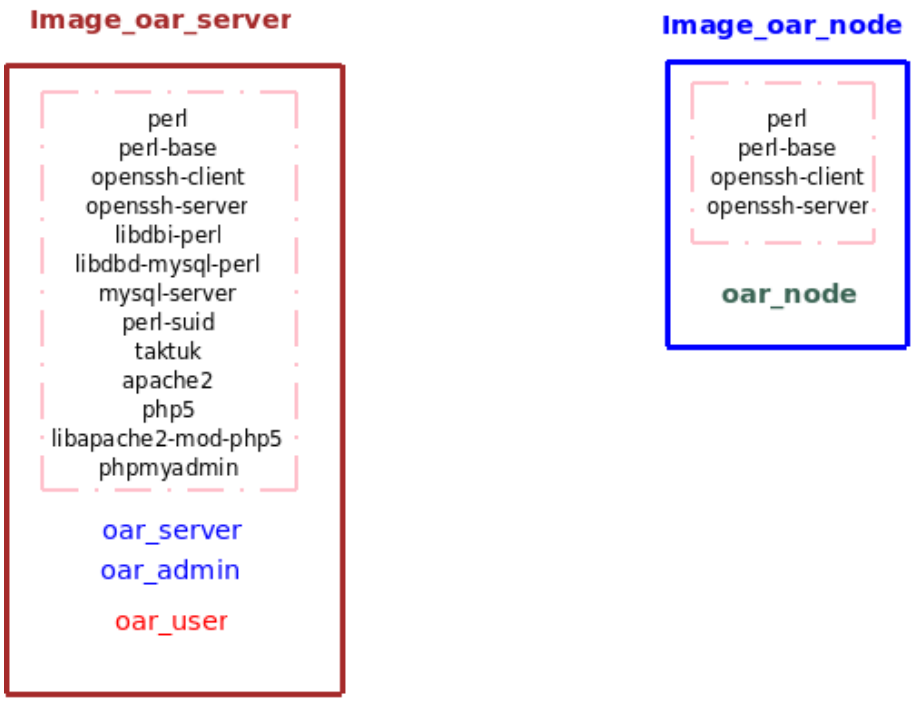

FIGURE 7.1 – Construction des images

est monté en synchrone sur les nœuds de calcul. Ceci permet de faire fonctionner les nœuds de calcul en mode *sans disque* comme nous l'avons adopté à la section 2.5. Toutes les opérations de lecture et écriture à partir d'un nœud de calcul sont donc effectuées dans le dossier */home*. Après la mise en place de cette configuration, la soumission des tâches peut alors commencer. La section suivante décrit la suite du scénario des expérimentations.

7.2 Scénario des expérimentations

Dans cette section, nous présentons comment nous avons simulé la présence des utilisateurs ainsi que la soumission de leurs travaux et l'exécution de ceux-ci.

7.2.1 Simulation de la présence des utilisateurs

Nous avons considéré à la section 6.3.1 que la soumission des tâches se fait à travers des réservations en mode passif. Dans ce cas les résultats des exécutions sont sauvegardés dans le fichier *OAR.JOB_ID.stdout* et les sorties d'erreur dans le fichier *OAR.JOB_ID.stderr* dans le dossier à partir duquel la tâche a été lancée.

Pour simuler la soumission des tâches par plusieurs utilisateurs distincts, dans le */home* du serveur, nous avons créé des répertoires $USER1$, $USER2$,...,$USERn$ dans lesquels les applica-

tions des utilisateurs ont été placées. Comme applications des utilisateurs, nous avons utilisé des benchmarks *Bench1*, *Bench2*, *Bench3*, *Bench4*. Cette organisation par répertoire des utilisateurs revêt deux aspects importants. Tout d'abord cela permet de sauvegarder les points de reprise de chaque benchmark exécuté dans le dossier de l'utilisateur correspondant. De plus cela permet d'éviter des situations où un point de reprise peut être écrasé par un autre. En effet de telles situations peuvent se produire si les travaux des utilisateurs étaient placés dans le même répertoire. Puisque les tâche des utilisateurs sont exécutées sur des nœuds différents, il n'est pas exclus que deux tâches aient le même PID. Étant donné que BLCR sauvegarde les points de reprise dans les fichiers *context.PID*, il est possible d'avoir cette situation où le premier fichier *context.PID* soit écrasé par le second de même nom.

7.2.2 Soumission et exécution des tâches

Avec la configuration décrite précédemment, nous avons écrit un script qui est chargé d'effectuer la soumission des tâches au gestionnaire de ressources en mode passif. Le script parcourt chaque dossier utilisateur et exécute la ligne de commande avec le benchmark indiqué. La ligne de commande soumise à OAR comprend le nombre de nœuds sollicités, la durée de l'exécution de la tâche et la commande utilisateur à exécuter sur les nœuds. Pour nos expériences, nous utilisons un nœud pour une commande utilisateur. La commande utilisateur contient le module client de l'ordonnanceur des sauvegarde *schedule-ckpt-c* qui prend en entrée le benchmark à exécuter (ou la liste des benchmarks dans le cas d'une exécution multiple sur un nœud). La syntaxe de soumission a été présentée à la section 6.4.1. Pour marquer le fait que les tâches sont soumises à des instants distincts, le script effectue une pause d'une durée déterminée, après chaque soumission.

En fonction des ressources disponibles, le gestionnaire des ressources exécute les tâches sur les nœuds demandés. Les requêtes qui n'ont pas pu être exécutées faute de ressources sont mises dans une file d'attente.

La section qui suit présente et analyse les résultats obtenus.

7.3 Résultats et discussions

Dans la suite, nous notons $TCCAS$ le temps cumulé de calcul des applications sauvegardées, $TCCANS$ le temps cumulé de calcul des applications non sauvegardées, *nb_ckpt* le nombre d'applications sauvegardées et *sche_time* le temps consommé par l'algorithme d'ordonnancement des sauvegardes $ScheduleCkpt$.

Au cours des tests, des mesures ont été faites et les données ont été sauvegardées dans des fichiers de trace en vue des analyses et discussions.

Pour les expériences, deux critères ont été considérés : sélectionner les applications dans l'ordre décroissant des temps de calcul non encore sauvegardé p_i ; sélectionner les applications dans l'ordre décroissant des densités p_i/s_i représentant le rapport entre le temps de calcul non

encore sauvegardé et la mémoire allouée.

7.3.1 Efficacité de la sauvegarde en parallèle

Cette section compare les résultats de l'implémentation du modèle d'ordonnancement que nous avons proposé à ceux obtenus avec une implémentation gloutonne naïve qui consiste à sauvegarder les processus séquentiellement dans le délai imposé. Les applications sont choisies suivant les p_i décroissants.

Les tableaux 7.1 et 7.2 présentent les résultats des expériences menées avec les paramètres $n = 50$ et $T = 300$. Les applications ont des tailles identiques, c'est-à-dire qu'elles occupent le même volume mémoire allouée. Nous avons fait varier les tailles de 10 Mo à 400 Mo. L'algorithme d'ordonnancement *ScheduleCkpt* a été exécuté avec $k_0 = 0$.

$n = 50 : T = 300 : P = \{26, 27, ..., 75\}$			
Taille (Mo)	nb ckpt	TCCAS (s)	TCCANS (s)
10	50	2525	0
50	50	2525	0
100	50	2525	0
200	45	2385	140
400	28	1722	803

TABLE 7.1 – Sauvegarde en parallèle avec *ScheduleCkpt*

$n = 50 : T = 300 : P = \{26, 27, ..., 75\}$			
Taille (Mo)	nb ckpt	TCCAS (s)	TCCANS (s)
10	50	2525	0
50	50	2525	0
100	16	1080	1445
200	8	572	1953
400	4	297	2228

TABLE 7.2 – Sauvegarde séquentielle gloutonne

Il ressort clairement des tableaux 7.1 et 7.2 que la sauvegarde en parallèle est meilleure que la sauvegarde séquentielle. En effet la sauvegarde en parallèle permet de réaliser de meilleures valeurs de *TCCAS*, ce qui indique une meilleure consommation des ressources.

Les résultats des tests qui suivent présentent la performance de l'algorithme d'ordonnancement implémenté.

7.3.2 Performance de l'ordonnanceur des sauvegardes

Au cours de nos expériences, nous avons noté que pour $k_0 \geq 3$, le temps d'exécution de l'algorithme d'ordonnancement était du même ordre que celui du délai imposé T, ce qui n'est

pas raisonnable. En effet, l'approche de Sartaj tend à être exponentiel pour $k_0 \geq 3$ en fonction de la taille de l'instance traitée. Par conséquent, dans la suite, nous considérons $k_0 < 3$.

Dans les tests qui suivent, nous avons mené un grand nombre de tests sur différents jeux de données répartis en 3 classes de taille $n = 25$, $n = 50$ et $n = 70$. Les résultats obtenus dans chaque classe étaient presque similaires. Dans la suite, nous ne présentons que les résultats qui illustrent mieux les performances de l'algorithme d'ordonnancement, dans chaque classe. Pour les 3 classes considérées, les tailles en mémoire s_i des applications se situent entre 100 Mo et 500 Mo.

Nous avons utilisé l'implémentation de l'architecture étendue de l'ordonnanceur des sauvegardes, avec la technique de sauvegarde par vague qui permet d'exploiter la puissance de calcul des nœuds qui sont de plus en plus multiprocesseurs multicœurs. Nous présentons les résultats des tests dans le cas où on a une application par nœud en cours d'exécution et dans le cas où plusieurs applications par nœud s'exécutent en parallèle.

Cas de la sauvegarde avec une application par nœud

Dans cette partie chaque nœud de calcul exécute une application. Les tests ont été effectués sur la grappe Azur de Sophia sur des nœuds monoprocesseur monocœur.

Le tableau 7.3 donne les performances de *ScheduleCkpt* avec le critère p_i et le tableau 7.4 avec le critère p_i/s_i. Pour ces tests, $n = 25$, $T = 180$s (3 minutes) et les temps de calcul non encore sauvegardé p_i sont compris entre 60 s (1 minute) et 600 s (10 minutes).

ScheduleCkpt	$n = 25 : T = 180(s)$		
	TCCAS (s)	TCCANS (s)	sche_time (μs)
$k_0 = 0$	4275	6266	30
$k_0 = 1$	6134	4407	1825
$k_0 = 2$	6987	3554	20984

TABLE 7.3 – Résultat de *ScheduleCkpt* pour le critère p_i, avec $n = 25$ et $T = 180(s)$

ScheduleCkpt	$n = 25 : T = 180(s)$		
	TCCAS (s)	TCCANS (s)	sche_time (μs)
$k_0 = 0$	5537	5004	32
$k_0 = 1$	6992	3549	2498
$k_0 = 2$	7322	3219	22005

TABLE 7.4 – Résultat de *ScheduleCkpt* pour le critère p_i/s_i, avec $n = 25$ et $T = 180(s)$

Le tableau 7.5 donne les performances de *ScheduleCkpt* avec le critère p_i et le tableau 7.6 avec le critère p_i/s_i. Pour ces tests, $n = 50$, $T = 300$s (5 minutes) et les temps de calcul non encore sauvegardé p_i sont compris entre 300 s (5 minutes) et 1800 s (30 minutes).

ScheduleCkpt	$n = 50 : T = 300(s)$		
	TCCAS (s)	TCCANS (s)	sche_time (μs)
$k_0 = 0$	18734	12678	83
$k_0 = 1$	23251	8161	6421
$k_0 = 2$	26128	5284	72417

TABLE 7.5 – Résultat de *ScheduleCkpt* pour le critère p_i, avec $n = 50$ et $T = 300(s)$

ScheduleCkpt	$n = 50 : T = 300(s)$		
	TCCAS (s)	TCCANS (s)	sche_time (μs)
$k_0 = 0$	20891	10521	152
$k_0 = 1$	24516	6896	5612
$k_0 = 2$	28874	2538	65589

TABLE 7.6 – Résultat de *ScheduleCkpt* pour le critère p_i/s_i, avec $n = 50$ et $T = 300(s)$

Le tableau 7.7 donne les performances de *ScheduleCkpt* avec le critère p_i et le tableau 7.8 avec le critère p_i/s_i. Pour ces tests, $n = 70$, $T = 420$s (7 minutes) et les temps de calcul non encore sauvegardé p_i sont compris entre 300 s (5 minute) et 1800 s (30 minutes).

ScheduleCkpt	$n = 70 : T = 420(s)$		
	TCCAS (s)	TCCANS (s)	sche_time (μs)
$k_0 = 0$	14956	27001	169
$k_0 = 1$	23765	18192	51674
$k_0 = 2$	25897	16060	265782

TABLE 7.7 – Résultat de *ScheduleCkpt* pour le critère p_i, avec $n = 70$ et $T = 420(s)$

ScheduleCkpt	$n = 70 : T = 420(s)$		
	TCCAS (s)	TCCANS (s)	sche_time (μs)
$k_0 = 0$	15984	25973	227
$k_0 = 1$	24143	17814	38977
$k_0 = 2$	27089	14864	290314

TABLE 7.8 – Résultat de *ScheduleCkpt* pour le critère p_i/s_i, avec $n = 70$ et $T = 420(s)$

Les tableaux 7.3, 7.4, 7.5, 7.6, 7.7 et 7.8 montrent que la qualité de l'algorithme d'ordonnancement croît avec k_0. En effet *TCCAS* est de plus en plus élevé, ce qui marque une meilleure utilisation des ressources. Nous pouvons également noter que le temps consommé par l'algorithme d'ordonnancement pour son exécution croît aussi avec k_0 mais demeure négligeable comparé au délai imposé T. Les résultats montrent par ailleurs que le critère de choix p_i/s_i produit de meilleures performances au vu des valeurs de *TCCAS*.

Cas de la sauvegarde avec plusieurs applications par nœud

Pour ces tests, chaque nœud exécute en parallèle 4 applications. Les tests ont été effectués sur la grappe Azur de Sophia sur $n = 25$ nœuds biprocesseurs bicœurs. Ainsi 100 applications étaient candidates à la sauvegarde. Nous n'avons considéré que le cas $k_0 = 2$ qui produit de meilleures solutions. $T = 600$s (10 minutes) et les temps de calcul non encore sauvegardé p_i sont compris entre 300 s (5 minute) et 1800 s (30 minutes).

Le tableau 7.9 donne les performances de *ScheduleCkpt* avec le critère p_i et le tableau 7.10 avec le critère p_i/s_i.

ScheduleCkpt	$n = 25 : T = 600(s)$		
	TCCAS (s)	TCCANS (s)	sche_time (μs)
$k_0 = 2$	27084	13132	92104

TABLE 7.9 – Performance de *ScheduleCkpt* pour la sauvegarde en parallèle par vague avec le critère p_i

ScheduleCkpt	$n = 25 : T = 600(s)$		
	TCCAS (s)	TCCANS (s)	sche_time (μs)
$k_0 = 2$	29217	10999	97589

TABLE 7.10 – Performance de *ScheduleCkpt* pour la sauvegarde en parallèle par vague avec le critère p_i/s_i

Même dans le cas de la sauvegarde par vague avec plusieurs applications en cours d'exécution par nœud, on note que le temps d'exécution de l'algorithme d'ordonnancement demeure négligeable, comparé au délai imposé T.

7.4 Conclusion

Ce chapitre a eu pour objectif de présenter les performances du modèle d'ordonnancement des sauvegardes proposé dans cette thèse. Les tests ont été effectués à partir de l'implémentation de l'architecture étendue avec approche de sauvegarde par vague. Les résultats obtenus montrent que l'algorithme d'ordonnancement n'induit pas un surcoût considérable sur les performances des mécanismes du checkpointing, que ce soit dans le cas de la sauvegarde avec une application par nœud que dans le cas de la sauvegarde par vague avec plusieurs applications par nœud.

Chapitre 8

Conclusion et perspectives

8.1 Rappel des objectifs

Dans cette thèse, nous avons considéré un contexte où les ressources disponibles d'un intranet sont utilisées comme une grappe virtuelle pour le calcul scientifique au cours de périodes d'inactivité comme les nuits, les week-ends et les périodes de congé. Généralement, ces périodes d'inactivité ne permettent pas toujours de réaliser les calculs jusqu'à leur terme. Il est donc nécessaire de sauvegarder le contexte d'exécution des applications inachevées en vue d'une éventuelle reprise, mais il n'est pas toujours possible de les sauvegarder toutes compte tenue des contraintes de temps. En supposant que ces applications sont indépendantes, nous sommes confrontés au problème de la sauvegarde d'un sous-ensemble parmi n applications en cours d'exécution sur les postes de travail qui doivent être libérés avant un délai T, tout en maximisant le temps agrégé de calcul non encore sauvegardé.

Notre objectif était de proposer un modèle d'ordonnancement des sauvegardes d'applications, afin d'utiliser efficacement les ressources libres des intranets comme infrastructure de calcul scientifique.

8.2 Démarche élaborée

La démarche proposée pour réaliser nos objectifs s'articule suivant trois parties.

La première positionne le problème de la sauvegarde d'applications de calcul haute performance dans les environnements dynamiques. Elle introduit les principales approches d'exploitation des ressources libres des intranets pour en faire des grappes de calcul. L'étude des mécanismes de sauvegarde/reprise d'applications, des différentes techniques utilisées et quelques systèmes implémentant ces techniques a permis de présenter certains paramètres importants à prendre en compte pour étude des performances des mécanismes de sauvegarde/reprise. L'analyse des différentes stratégies de sauvegarde conduit à l'adoption de la sauvegarde en parallèle qui cadre avec le souci d'utiliser à bon escient les ressources libres d'un intranet. Ce choix

positionne en même temps le problème des goulets d'étranglement qui en découlent.

La seconde est dédiée à la proposition d'un modèle d'ordonnancement des sauvegardes d'applications. Elle étudie dans un premier temps le problème de congestion réseau et disque lié à la sauvegarde en parallèle. Cette analyse est réalisée à partir d'un ensemble d'expérimentations qui ont pour but d'estimer la bande passante utilisée. La finalité de cette prédiction est de pouvoir élaborer un modèle de sauvegarde sans perte de performance du dispositif de sauvegarde, tout en maximisant la consommation des ressources.

La dernière partie présente l'intégration d'une implémentation du modèle d'ordonnancement des sauvegardes d'applications proposé dans un gestionnaire de ressources et évalue les performances du modèle. Les différents éléments d'implémentation et le fonctionnement du gestionnaire de ressources avec l'ordonnanceur des sauvegardes dans une grappe sont décrits. L'évaluation est faite dans un environnement réel construit sur la plateforme GRID5OOO de façon à reproduire le fonctionnement d'un intranet en mode infrastructure de calcul scientifique.

8.3 Travaux réalisés

sur la base d'un grand nombre d'expérimentations, nous avons proposé au chapitre 4, une fonction qui donne la bande passante du système nécessaire pour la sauvegarde en parallèle d'un certain nombre d'applications séquentielles indépendantes et de leur volume mémoire agrégé. Cette fonction est utilisée pour estimer le temps de sauvegarde de chacune des applications, dans un contexte où la bande passante est équitablement partagée entre les différentes tâches. Cette fonction est utilisée pour calibrer le nombre d'applications que l'on peut sauvegarder sans perte de performance du dispositif de sauvegarde. Cet estimateur permet de prédire la bande passante utilisée avec une erreur faible d'environ 4%. L'étude menée pour le cas de la sauvegarde d'applications parallèles à révélé la grande difficulté à établir un estimateur de la bande passante utilisée, compte tenu de certains paramètres non déterministes comme la présence des messages dans les canaux de communications. Ceci nous a amené à restreindre la suite des travaux au cas des applications séquentielles indépendantes.

Pour le problème du choix des applications à sauvegarder dans un certain délai tout en maximisant la consommation des ressources, nous avons proposé au chapitre 5, une formulation toute nouvelle d'un problème d'optimisation basée sur l'approche du problème du sac-à-dos. La preuve de la NP-complétude de cette nouvelle classe de problème d'optimisation a été donnée. Ensuite nous avons proposé un algorithme d'ordonnancement des sauvegardes des applications où la sélection des candidats se fait en deux phases. La première utilise la fonction d'estimation de la bande passante pour déterminer avec une approche gloutonne, les candidats qui peuvent être sauvegardés en parallèles sans perte de performance du système. La seconde utilise l'approche semi-énumérative présentée dans [8] pour déterminer les tâches à sauvegarder effectivement en tenant compte de la contrainte délai et choisit la meilleure solution.

Pour valider le modèle d'ordonnancement des sauvegardes proposé, au chapitre 6 nous avons présenté son intégration dans un gestionnaire de ressources. Ce modèle a été implémenté et intégré au gestionnaire de ressources OAR comme un module indépendant. Le prototype im-

plémenté est une application client/serveur (basée sur le modèle *un thread par client accepté* qui exploite les données provenant de la base de données du gestionnaire de ressources de la grappe pour faire l'ordonnancement des sauvegardes. Par exemple, pour le cas de l'intégration à OAR, l'interaction entre le prototype implémenté de l'ordonnanceur des sauvegardes avec la base de données de OAR a été présentée. Ce prototype est portable et peut être intégré à tout gestionnaire de ressources, du moment que la structure de la base de données est connue et qu'il existe un estimateur de la bande passante du système. Les tests de validation menés sur Grid5000 ont montré la pertinence de la sauvegarde en parallèle. Les résultats obtenus ont également montré que le temps d'exécution de l'algorithme d'ordonnancement est négligeable par rapport au délai T, que ce soit dans le cas de la sauvegarde avec une application par nœud que dans le cas de la sauvegarde par vague avec plusieurs applications par nœud. Ceci signifie que l'algorithme d'ordonnancement proposé n'induit pas un surcoût significatif sur les mécanismes de sauvegarde.

8.4 Perspectives

Cette thèse soulève d'intéressantes questions qui constituent un champ d'investigation à explorer.

8.4.1 Signification des coefficients de bw

Compte tenu de la complexité qui s'est dégagée de l'analyse technique que nous avons menée sur les composantes du chemin de sauvegarde, nous avons proposé d'étudier de manière expérimentale le problème de la sauvegarde en parallèle d'applications vers le disque du serveur de l'intranet. Pour cela, nous avons considéré que la soumission des sauvegardes des applications est effectuée à travers une boîte noire qui encapsule toutes les composantes du chemin de sauvegarde. Cette démarche nous a amené à observer l'évolution de la bande passante de la boîte noire, c'est-à-dire la bande passante agrégée des bandes passantes individuelles. Au vu des expériences menées, nous avons approximé cette bande passante par une fonction polynômiale du second degré bw qui dépend de deux paramètres : m le nombre d'applications et V leur taille agrégé.

Cependant, la signification des coefficients de bw reste indéterminée. Une étude de la signification de ces coefficients pourrait permettre de mieux cerner le problème de congestion réseau et disque qui se pose lors de la sauvegardes d'applications vers un serveur unique.

8.4.2 Généralisation de l'expression de bw

L'expression de bw comme une fonction polynômiale du second degré à deux variables offre d'importantes propriétés. Par exemple dans l'environnement expérimental où nos études ont été menées, lorsque l'on fixe un des paramètres, la courbe de bw est concave. Cela permet d'obtenir de façon prédictive les pics au delà desquels la bande passante du système se dégrade. Il serait

donc intéressant de déterminer l'expression de *bw* dans d'autres plateformes avec différentes performances de réseaux et de voir ensuite s'il existe une formule polynômiale générale pour ces fonctions. Un tel résultat pourrait permettre par exemple de mettre en place une bibliothèque qui serait utilisée pour déterminer de façon automatique les coefficients de *bw* sur tout type de plateforme.

8.4.3 Algorithmes efficaces pour la nouvelle classe de problèmes du sac-à-dos formulée

Nous avons présenté une formulation du problème de la sauvegarde en parallèle d'applications, avec des contraintes de bandes passantes et une contrainte temporelle, dans un environnement dynamique, comme une variante du problème du sac-à-dos 0/1 où la contrainte est non linéaire. De la formulation détaillée du problème, nous avons proposé un algorithme basé sur une séquence de résolution du problème du sac-à-dos. Une question intéressante porte sur la conception d'algorithmes efficaces pour cette nouvelle classe de problèmes du sac-à-dos où les contraintes sont non linéaires.

8.4.4 Sauvegarde sur plusieurs serveurs

Cette question concerne la sauvegarde des points de reprise sur plusieurs serveurs. En effet, sauvegarder les points de reprise sur un unique serveur, comme nous l'avons considéré dans cette thèse, pourrait induire des pertes de performance dûes à des goulots d'étranglement et un unique point de défaillance. Une solution au problème de congestion a été proposée dans ce travail. Cependant, en cas de crash du serveur tout le système est paralysé, du moins par rapport au redémarrage des points de reprise. Le problème de la sauvegarde des points de reprise sur plusieurs serveurs est donc une question importante, car cela permet à l'environnement distribué d'être plus tolérant aux pannes. Suivant l'approche introduite dans cette thèse, ce problème pourrait nécessiter, par exemple dans le cas particulier de deux serveurs, la construction de la fonction $(m_1, V_1, m_2, V_2) \longmapsto bw(m_1, V_1, m_2, V_2)$ qui donne la bande passante du système lorsque $m_1 + m_2$ tâches de volume mémoire agrégé $V_1 + V_2$ sont sauvegardées en parallèle, m_1 sur le premier serveur et m_2 sur l'autre.

8.4.5 Sauvegarde en parallèle complète avec plusieurs applications par nœud

Dans l'optique d'exploiter efficacement la puissance de calcul délivrée par les nœuds qui sont de plus en plus multiprocesseurs multicœurs, nous avons proposé une extension de l'architecture de l'odonnanceur des sauvegardes. La nouvelle architecture permet d'exécuter plusieurs applications par nœud, en les préparant à la sauvegarde. Pour la sauvegarde des applications nous avons proposé deux approches. La première permet d'exploiter le modèle d'ordonnancement proposé avec une application par nœud, pour effectuer des sauvegardes par vague, en fonction

du délai imposé. Cette approche a été implémentée et validée. Cependant, elle peut entraîner une sous-utilisation de la bande passante du système, par exemple dans le cas où le nombre de processus dans une vague est inférieur à celui supportable par la bande passante. La seconde propose d'exploiter entièrement le parallélisme offert au sein d'un nœud pour la sauvegarde en parallèle d'applications. Cependant, outre les conflits inter-nœuds liés à la sauvegarde en parallèle, cette approche induit également des conflits intra-nœud lors de la sauvegarde en parallèle de plusieurs applications s'exécutant sur le nœud. Une étude de ces conflits serait intéressante, car cette approche n'induit pas de sous-utilisation de la bande passante du système.

Annexes

Annexe A

Détails sur la procédure de sélection des applications à sauvegarder

Cette annexe présente en détail la procédure *ks_bw* de l'algorithme d'ordonnancement des sauvegardes présenté au chapitre 5. *ks_bw* est l'algorithme de Sartaj [8] modifié et adapté à notre problème de sélection des applications candidates qui doivent être effectivement sauvegardées. Les notations sont celles présentées dans le chapitre 5.

La procédure *ks_bw* utilise la procédure L suivante pour compléter de façon gloutonne l'ensemble $Select$ des candidats déjà sélectionnés avec les candidats restants $i \in I - Select$ suivant le critère choisi (valeurs décroissantes des p_i ou des p_i/s_i), et les insère dans $Select$ si $(\max_{j \in Select \cup \{i\}} s_j)/[bw(m, \sum_{j \in Select \cup \{i\}} s_j)/m] \leq T - t_{ckpt}$, où $m = |Select \cup \{i\}|$. La procédure L est l'algorithme 12 suivant :

Algorithme 3 : $L(I, Select, S', T - t_{ckpt})$

```
1  Init s_ckpt ← Σ_{k∈Select} s_k ; L ← 0 ; np ← Card(Select) ;
2  for each i ∈ I − Select do
3  |   s_max ← max{s'_i, s'_{j,j∈Select}} ;
4  |   if s_max/[bw(np + 1, s_ckpt + s'_i)/(np + 1)] ≤ T − t_ckpt then
5  |   |   Select ← Select ∪ {i} ;
6  |   |   np ← np + 1 ;
7  |   |   s_ckpt ← s_ckpt + s'_i ;
8  |   |   L ← L + p_i ;
9  |   end
10 end
11 Sack ← Select ;
12 return (L) ;
```

Décrivons maintenant la procédure *ks_bw*. Considérer tous les sous-ensembles $J \subseteq I - CKPT$ de taille k_0 et choisir ceux qui satisfont à la condition
$t_J = (\max_{\{i \in J \cup CKPT\}} s_i)/[bw(m, \sum_{i \in J \cup CKPT} s_i)/m] \leq T$, où $m = |J \cup CKPT|$.

Considérer que les blocs d'un ensemble J et ceux des applications en cours de sauvegarde sont sélectionnés, c'est-à-dire $Select := J \cup CKPT$. La procédure ks_bw est l'algorithme 7 suivant

Algorithme 4 : $ks\_bw(I, E', S', CKPT, T - t_{ckpt}, k_0)$

1 *// E' dénote l'ensemble des processus dont les tailles sont indexées dans S' ;*
2 **Init** $PMAX \leftarrow 0$;
3 **for** ***all*** *combinations $J \subseteq I - CKPT$ of size k_0 and $t_J \leq T - t_{ckpt}$* **do**
4 $\quad Select \leftarrow J \cup CKPT$;
5 $\quad P_J \leftarrow \sum_{j \in J} p_j$;
6 $\quad PMAX \leftarrow \max\{PMAX, P_J + L(I, Select, S', T - t_{ckpt})\}$;
7 **end**

A la fin de l'exécution de ks_bw, l'ensemble $Select^*$ qui maximise $\sum_{j \in Select} p_i$ est récupéré dans le tableau $Sack$.
Les applications qui doivent être nouvellement sauvegardés sont ceux de $Sack - CKPT$.

Annexe B

Installation des services sur des machines virtuelles

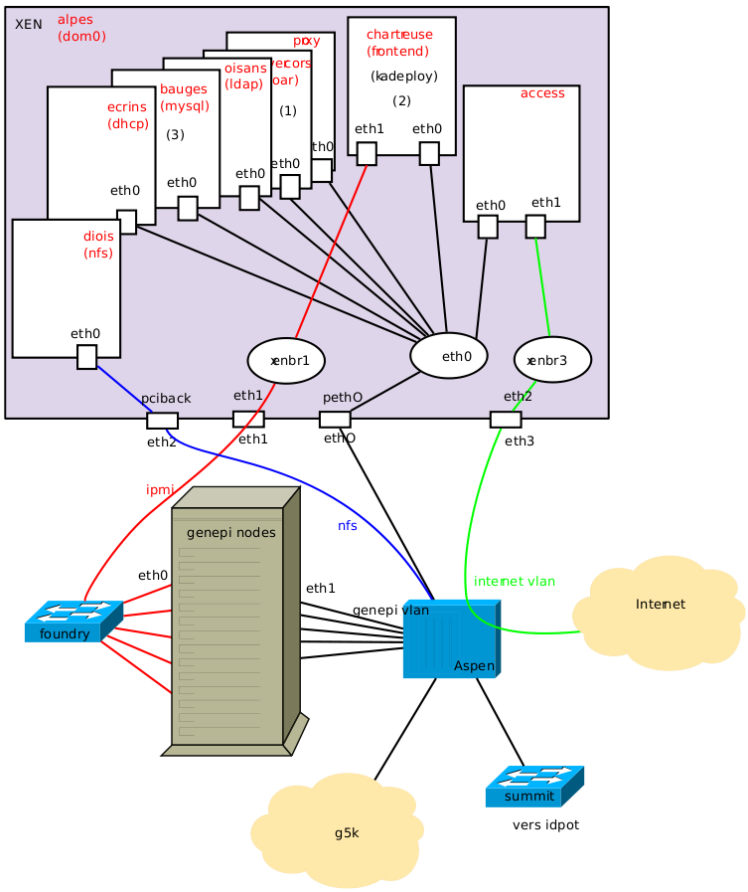

FIGURE B.1 – Placement des services sur des machines virtuelles au niveau du serveur d'une grappe de Grid5000 (*source : https ://gforge.inria.fr/projetcs/grid5000*)

Annexe C

Schéma de la base de données de OAR

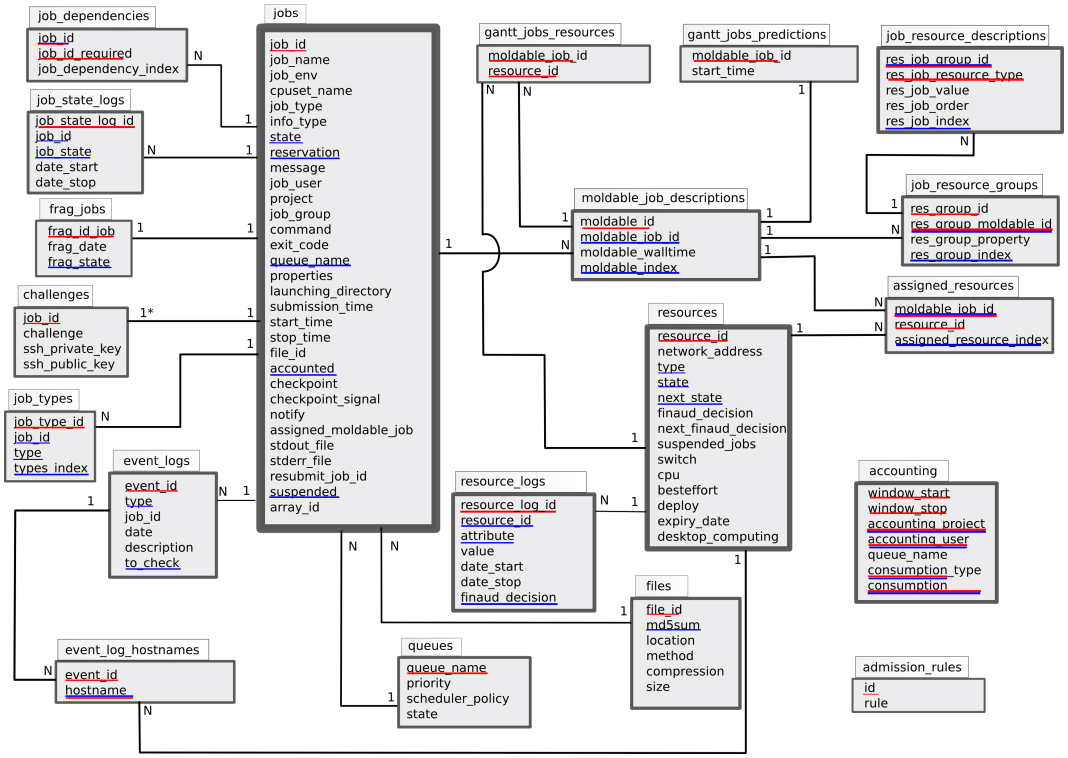

FIGURE C.1 – Schéma de la BD de OAR (*source : https ://gforge.inria.fr/projetcs/grid5000*)

Bibliographie

[1] F. Dupros, F. Boulahya, J. Vairon, P. Lombard, N. Capit, and J.-F. Mehaut. IGGI, a computing framework for large scale parametric simulations : application to uncertainty analysis with toughreact. In *Proceedings, Tough Symposium*, 2006.

[2] K. M. Chandy and L. Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[3] Antonio Cueni and Jan Vitek. A New Approach To Real-Time Checkpointing. In *VEE'06, Ottawa, Ontario, Canada*, June 14-26, 2006.

[4] E. N. Elnozahi, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proc. 11th IEEE Symp. on Reliable Distributed Systems*, pages 39–47, October, 1992.

[5] M. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June, 1988.

[6] J.M. Nlong II and Y. Denneulin. Migration des processus Linux sous I-cluster. In *RENPAR'15 / CFSE'3 / SympAAA'2003. La Colle sur Loup, France*, 15 au 17 octobre 2003.

[7] J.W. Young. A First Order Approximation to the Optimum Checkpoint Interval. *Communcations of the ACM*, 17(9) :530–531, September 1974.

[8] Sartaj Sahni. Approximate Algorithms for the 0/1 Knapsack Problem.

[9] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Proceedings of Cluster Computing and Grid (CCGRID)*, May 2005.

[10] W. Richard Stevens. *Unix Network Programming*, volume 1. Prentice Hall PTR, second edition, 1998.

[11] B. O. Yenke. Prédiction des performances des opérations de sauvegarde/reprise sur cluster virtuel. In *Actes RENPAR'18 / SympAAA'2008 / CFSE'3 / Fribourg, Suisse*, du 11 au 13 février 2008.

[12] B. O. Yenke and M. Mehaut, J-F. andTchuente. Scheduling deadline-constrained checkpointing on virtual clusters. In *Proceedings of the IEEE Asia-Pacific Services Computing Conference, Yilan, Taiwan*, pages 257–264, December 9-12, 2008.

[13] B. O. Yenke and M. Mehaut, J-F. andTchuente. Scheduling of Computing Services on Intranet Networks. *IEEE Transaction on Service Computing*, Accepted for publication, to appear 2010.

[14] B. O. Yenke, J-F. Mehaut, J. M. Nlong II, and R. Chakode. Integrating deadline-constrained checkpointing in a batch scheduler for dynamic environments. In *Proceedings of the Annual International Conference on Software Engineering (SE 2010), Phuket Beach Resort, Thailand*, pages S–156 – S–163, December 6-7, 2010.

[15] MESCAL. Inria project : Calcul distribué et applications haute performance, 2005. http ://www.inria.org/recherche/equipes/mescal.fr.html.

[16] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9) :948–960, 1972.

[17] A. S. Tanenbaum and van Maarten Steen. *Distributed Systems. Principles and Paradigms.* Prentice Hall, Upper Saddle River, New Jersey 07458, 2002.

[18] Jack Dongarra, Dennis Gannon, Geoffrey Fox, and Ken Kennedy. The Impact of Multicore on Computational Science Software. *CTWatch Quarterly*, 3(1), February 2007.

[19] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf : A Parallel Workstation For Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[20] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. *International J. Supercomputer Applications, 15(3)*, 2001.

[21] F. Capello and al. Grid'5000 : a large scale, reconfigurable, controlable and monitorable grid platform. In *Proceedings of IEEE/ACM Grid'2005 workshop*, 2005.

[22] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds : Towards a Cloud Definition. *ACM SIGCOMM Comput. Commun. Rev.*, 39(1) :50–55, 2009.

[23] F. Capello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Neri, and O. Lodygensky. Computing on Large Scale Distributed Systems : XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. . *FGCS Future Generation Computer Science*, 2004.

[24] Gilles Fedak, Cecile Germain, Vincent Neri, and Franck Capello. XtremWeb : A generic global computing system. In *IEEE CCGrid2001 (Global Computing on Personal Devices)*, 2001.

[25] D. Anderson and al. SETI@home : The Search for Extraterrestrial Intelligence. Technical report, University of California at Berkeley, 1999.

[26] S. M. Larson, C. D. Snow, M. R. Shirts, and V. S. Pande. Folding@Home and Genome@Home : Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2003.

[27] B. Richard and P. Augerat. Effective Aggregation of Idle Computing Resources for Cluster Computing. *Journal of European Research Consortium for Informatics and Mathematics (ERCIM)*, October 2004.

[28] N. Capit, L. Desbat, L. Eyrauld, and O. Richard. CIMENT GRID : a GRID facility for large scale parametric computing. In *Workshop PMAA*, October 2004.

[29] W. Gropp, E. Lusk, and T. Sterling. *Beowulf Cluster Computing with Linux, Second Edition*. The MIT Press, Cambridge, Massachusetts, London, England, 2003.

[30] C. Eddy and U. Gil. On the Performance of Parallel Factorization of Out-of-Core Matrices. *Parallel Computing, 30(3) :357-375*, February 2004.

[31] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen king Su. Myrinet - A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15 :29–36, 1995.

[32] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The quadrics network : High performance clustering technology. *IEEE Micro*, 22(1) :46–57, January 2002.

[33] InfiniBand. InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 2000. http ://www.ifinibandta.org.

[34] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 : Desing and implementation. In *Proceedings of Summer 1994 USENIX Conference*, pages 137–152, 1994.

[35] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Desing and implementation of the Sun network file system. In *Proc. of Usenix 1985 Summer, Portland*, pages 119–130, June 1985.

[36] Scmuck Frank and Haskin Roger. GPFS : A shared-disk file system for large computing clusters. In *First USENIX Conference on File and Storage Technologies (FAST'02), Monterey, CA*, January 28-30 2002.

[37] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PNFS : A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.

[38] A. Bayucan and R. L. Henderson. Portable Batch System. http ://pbs.mrj.com. Technical report, November 1999.

[39] R. L. Henderson. Job scheduling under the Portable Batch System. In *Job Scheduling Strategies for Parallel Processing. IPPS'95 Workshop Proceedings*, April 1995.

[40] David B. Jackson, Quinn Snell, and Mark J. Clement. Core Algorithms of the Maui Scheduler. In *JSSPP '01 : Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102, London, UK, 2001. Springer-Verlag.

[41] Wolfgang Gentzsch. Sun Grid Engine : Towards Creating a Compute Power Grid. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36, 2001.

[42] Liang PENG and Lip Kian NG. N1GE6 Checkpoingting and Berkeley Lab Checkpoint/-Restart. *Sun Microsystems*, Dec. 2004.

[43] Bruno Richard. *Agrégation des ressources inexploitées d'un intranet et exploitation pour l'instanciation de services de calcul intensif*. PhD thesis, Istitut National Polytechnique de Grenoble, Décembre 2003.

[44] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice : The Condor Experience. *Concurrency and Computation : Practice and Experience*, 17 :2–4, 2005.

[45] R. Figueiredo, P. Dinda, and J. Fortes. A Case for Grid Computing on Virtual Machines. In *23rd International Conference on Distributed Computing Systems*, 2003.

[46] K. Keahey, K. Doering, and I. Foster. From Sandbox to Playground : Dynamic Virtual Environments in the Grid. In *6th International Workshop in Grid Computing*, 2004.

[47] S. Adabala, V. Chadha, P. Chawla, R. Figueiredo, I. Fortes, J. ande Krsul, A. Matsunaga, J. Tsugawa, M. ande Zhang, M. Zhao, L. Zhu, and X. Zhu. From Virtualized Resources to Virtual Computing Grids : The In-VIGO System . *Future Generation Computer Science*, 2004.

[48] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Towards Virtual Distributed Environments in a Shared Infrastructure. *IEEE Computer, Special Issue on Virtualization Technologies*, 2005.

[49] I. Foster, T. Freeman, K. Keahy, D. Scheftner, B. Sotomayer, and X. Zhang. Virtual Clusters for Grid Communities. In *CCGRID '06 : Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, pages 513–520, 2006.

[50] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[51] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualisable Third Generation Architectures. *Communcations of the ACM*, 17(7) :412–421, July 1974.

[52] L. Jiuxing and B. Abali. Virtualization Polling Engine (VPE) : Using Dedicated CPU Cores to Accelerate I/O Virtualization. In *Proceedings of the ICS'09, Yorktown Heights, New York, USA*, June 8-12 2009.

[53] A. Tikotekar and A. M. Filippi. Effects of Virtualization on Scientific Application. Running a Hyperspectral Transfert Code on Virtual Machines. In *Proceedings of the 2nd Workshop on System-level Virtualization for High Performance Computing (HPC-Viert'08), Glasgow, Scotland*, Mars 31 2008.

[54] A. Avizienis, J. C. Laprie, and B. Randell. Fundamental concepts of dependability. Technical report, Research Report N01145, LAAS-CNRS. Technical Report Research Report N01145, University of Wisconsin, Madison, 2001.

[55] Pankaj Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, Inc, Upper Saddle River, New Jersey 07458, 1998.

[56] J. Duell, P. Hargrove, and E. Roman. Requirements for Linux Checkpoint/Restart. Technical Report publication LBNL-49659, 2002.

[57] James S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, July 1997.

[58] M. Litzkow, T. Tannenbauw, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Process in the Condor Distributed Processing System. Technical Report CS-TR-1997-1346, University of Wisconsin, Madison, Apr. 1997.

[59] Elmootazbellah Elnozahy and James Plank. Checkpointing for peta-scale systems : A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2) :97–108, 2004.

[60] H. Higaki, K. Shima, T. Tachokawa, and M. Takizawa. Checkpoint and Rollback in Asynchronous Distributed Systems. *IEEE Transaction on Software Engineering*, May, 1997.

[61] T. T. Y. Juang and S. Venkatesan. Efficient Algorithm for Crash Recovery in Distributed Systems. In *10th Conference on Foundations of Software Technology and Theoretical Computer Science (LNCS)*, pages 349–361, 1990.

[62] T. T. Y. Juang and S. Venkatesan. Crash Recovery with Little Overhead. In *11th Conference on Distributed Computing Systems*, pages 454–461, 1991.

[63] J. M. Nlong II. *Conception et réalisation d'un intergiciel pour la résilience d'applications parallèles distribuées sur un intranet et Internet.* PhD thesis, Faculté des Sciences, Université de Yaoundé 1, Juillet 2008.

[64] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.*, 13(1) :23–31, 1987.

[65] Yanqing Ji, Hai Jiang, and Vipin Chaudhary. Adaptation Point Analysis for Computation Migration/Checkpointing. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC'05), Santa Fe, New Mexico, USA*, pages 750–751, 2005.

[66] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt : Transparent Checkpointing under Unix. In *Proceedings of the 1995 Winter USENIX Technical Conference*, 1995.

[67] Cryopid. CryoPID : A Process Freezer for Linux. http ://cryopid.berlios.de.

[68] Michael Rieker, Jason Ansel, and Gene Cooperman. Transparent user-level Checkpointing for Native POSIX Thread Library for Linux. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA-06)*, pages 492–498, 2006.

[69] TIS-Comitee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification.* Version 1.2, May 1995.

[70] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report Berkeley Lab LBNL-54941, November 2003.

[71] Paul Hargrove and Jason Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux cluster. *Journal of Physics Conference Series*, 46 :494–499, September 2006.

[72] G. Stellner. CoCheck : Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 526–531, April 1996.

[73] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++. In *ACM SIGOPS Operating System Review, New York, NY*, pages 90–99, April 2006.

[74] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance Evaluation of Adaptative MPI. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, Mars 2006.

[75] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Capello. MPICH-V Project : A Multiprotocol Automatic Fault-Tolerant MPI. *International Journal of High Performance Computing Applications*, 20(3) :319–333, 2006.

[76] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine. The LAM/MPI Checkpoint/-Restart Framework : System-Initiated Checkpointing. *Proceedings, LACSI Symposium*, October 2003.

[77] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barret, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodal. Open MPI : Goals, Concept, and Design of Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI User's Group Meeting, Budapest, Hungary*, pages 97–104, Sep. 2004.

[78] Oshua Hursey, Jeffrey M. Squyres, Timothy I. 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), IEEE Computer Society, Long Beach, CA, USA*, 2007.

[79] Joseph Ruscio, Michael Heffner, and Srinidhi Varadarajan. DejaVu : Transparent user-level checkpointing, migration and recovery for distributed systemss. In *IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[80] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP : Transparent Checkpointing for Cluster Computations and the Desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[81] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguezb, and Franck Cappello. Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI. In *Proceedings of the ACM/IEEE SC2006 Conference, Tampa, Florida, USA*, November 2006.

[82] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptative Incremental Checkpointing for Massively Parallel Systems. In *ICS'04, Saint Lao, France*, pages 277–286, June 26-July 1, 2004.

[83] T. H. Feng and E. A. Lee. Incremental Checkpointing With Application to Distributed Discret Event Simulation. In *Proceedings of the 2006 IEEE Winter Simulation Conference*, pages 1004–1011, 2006.

[84] J. H. Sangho, J Hong, and S. Y. Shin. Space-efficient Page-level Incremental Checkpointing. In *SAC'05, Santa Fe, New Mexico, USA*, pages 1558–1562, March 13-17, 2005.

[85] E. Imamagic, D. D. Zagar, and B. Radic. Checkpointing approach for computer clusters. In *Information and Intelligent Systems (IIS) Conference, Zagreb, Croatie*, 21-23 September,2005.

[86] J.S. Plank and M. G. Thomason. The Average Availability of Uniprocessor Systems, Revisited. Technical Report UT-CS-98-400, Departement of Computer Science, University of Tenesse, August 1998.

[87] J.S. Plank and M. G. Thomason. The Average Availability of Parallel checkpointing Systems and its Importance in Selecting Runtime Parameters. In *29th International Symposium on Fault-Tolerant Computing (FTCS-29), Madison, WI*, June 18, 1999.

[88] Mohamd-Slim Bouguerra, Thierry Gautier, Denis Trystram, and Jean-Marc Vincent. A flexible checkpoint/restart model in distributed systems. In *Proceedings of the International IEEE conference PPAM'09*. Springer, to appear 2010.

[89] Y. Ling, J. Mi, and X. Lin. A Variational Calculus Approach to Optimal Checkpoint Placement. *IEEE Transactions on Computers*, 50(07) :699, July 2001.

[90] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3) :303–312, 2006.

[91] R. Y. de Camargo, R. Cerqueira, and F. Kon. Strategies for Checkpoint Storage on Opportunistic Grids. *IEEE Distributed Systems Online*, 7(9), 2006.

[92] J.S. Plank, K. Li, and M. A. Puening. Diskless Checkpointing. *IEEE Trans. on Parallel and Distributed Systems*, 9(10) :972–986, 1998.

[93] M. O. Rabin. Efficient Dispersal of Information for Security, Load Balancing and Fault Taulerence. *J. ACM*, 36(2) :335–348, 1989.

[94] X. Ren, R. Eigenmann, and S. Bagchi. Failure-Aware Checkpointing in Fine-Grained Cycle Sharing Systems, June 25-29, 2007.

[95] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proc. of Dependable Systems and Networks (DSN'05)*, pages 336–345, 2005.

[96] Maxime Martinasso. *Analyse et Modélisation des Communications Concurrentes dans les Réseaux Haute Performance*. PhD thesis, Université Joseph Fourier, école doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique, Grenoble, Mai 2007. CIFRE Bull.

[97] T. Olivares, L. Orozco-Barbosa, F. Quiles, A. Garrido, and P.J. Garcia. Performance study on NFS over Myrinet-Based clusters for parallel multimedia applications. *TIC*, 2000.

[98] P. Poullet, P. Nuiro, and J.-F. Méhaut. Parallel multilevel method for incompressible flows. In *Parallel multilevel method for incompressible flows, American Univ. of Beirut, Lebanon*, January 2006.

[99] Benoit Claudel, Guillaume Huard, and Olivier Richard. TakTuk, adaptive deployment of remote executions. In *HPDC '09 : Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 91–100, New York, NY, USA, 2009. ACM.

[100] Martello, S. and Toth, P. *Knapsack Problems : Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

[101] R. Bellman. Some applications of the theory of dynamic programming - a review. *Operations Research*, 2 :275–288, 1954.

[102] G.B. Dantzig. Discrete variables extremum problems. *Operations Research*, 5 :266–277, 1957.

[103] P.J. Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13 :723–735, 1967.

[104] G. P. Ingargiola and J. F. Korsh. A reduction algorithm for zero-one single knapsack problems. *Management Science*, 20 :460–463, 1973.

[105] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of ACM*, 21 :277–292, 1974.

[106] Silvano Martello and Paolo Toth. A new algorithm for the 0-1 knapsack problem. *Management Science*, 34(5) :633–644, 1988.

[107] W. Loots and T. H. C. Smith. A parallel algorithm for the zero-one knapsack problem. *International Journal of Parallel Program*, 21(5) :313–348, 1992.

[108] M. Ohlsson, C. Peterson, and B. Soderberg. Neural Networks for optimization problems with inequality constraints : the knapsack problem. *Neural Computation*, 5(2) :331–339, 1993.

[109] F.T. Lin. Solving the knapsack problem with imprecise weight coefficients using genetic algorithms. *European Journal of Operational Research*, 185(1) :133–145, 2008.

[110] A. J. Bryant. Greedy, genetic, and greedy genetic algorithms for the quadratic knapsack problem. In *GECCO '05 : Proceedings of the 2005 conference on Genetic and evolutionary computation, Washington, DC, USA*, pages 607–614, June 25-29, 2005.

[111] R. Kumar, A.H. Joshi, K.K. Banka, and P.I. Rockett. Evolution of Hyperheuristics for the Biobjective 0/1 Knapsack Problem by Multiobjective Genetic Programming, July 12-16, 2008.

[112] S. Sahni. Some related problems from network flows, game theory and integer programming. In *Proceeding of the 13th IEEE Symposium on Switching and Automata Theory*, pages 130–138, Oct 1972.