Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées à grande échelle

Xavier Besseron

UNIVERSITÉ DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

Docteur de l'Université de Grenoble

Spécialité : « Informatique »

préparée au Laboratoire d'Informatique de Grenoble

dans le cadre de **l'École Doctorale « Mathématiques, Sciences et Technologies de l'Information, Informatique »**

préparée et soutenue publiquement par

Xavier Besseron

le 28 avril 2010

Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées à grande échelle

---

sous la direction de

Thierry Gautier et Denis Trystram

Jury

Franck Cappello	INRIA	Rapporteur
Christophe Cérin	LIPN	Rapporteur
Frédéric Desprez	INRIA	Examinateur
Jacques Mossière	INPG	Examinateur (président du jury)
Denis Trystram	INPG	Examinateur
Thierry Gautier	INRIA	Examinateur

Remerciements

Pour commencer, je voudrais adresser mes remerciements aux membres du jury. Je remercie les rapporteurs, Franck Cappello et Christophe Cérin, qui ont eu le courage et qui ont pris le temps de lire avec attention ce document. Je remercie Jacques Mossière d'avoir accepté de présider ce jury et Frédéric Desprez d'être venu assister à la soutenance.

Je tiens également à remercier mes directeurs de thèse Denis Trystram et Thierry Gautier. Je connais Thierry depuis bientôt 5 ans. J'ai travaillé avec lui durant plusieurs stages et il m'a ensuite fait le plaisir de m'encadrer sur cette thèse. C'est un chercheur remarquable du point de vue scientifique et pour ses connaissances en programmation, mais également pour ses conseils et ses qualités humaines. J'ai pris beaucoup de plaisir à travailler et discuter avec toi, tu m'as beaucoup appris. Les pauses au labo et les bières que nous avons bues ensemble étaient des instants très agréables. Je te remercie vivement.

J'ai eu la chance de réaliser ces travaux de recherche dans le cadre du Laboratoire d'Informatique de Grenoble et plus précisément dans l'ex-labo ID. J'y ai rencontré des personnes formidables, avec lesquelles il est plaisant de travailler. La procrastination étant une de mes activités favorites, j'y ai trouvé de nombreux adeptes, qu'ils soient thésards ou chefs (je ne citerai personne, mais je garde les preuves). Ainsi, je remercie d'une manière globale tous les membres du labo pour leurs contributions (en travail ou en divertissement) à l'aboutissement de mon doctorat. Beaucoup sont devenus des amis.

Je pense que je n'aurais pas réussi à surmonter ces dures années de thèse sans la présence de mes amis. Ils m'ont permis de me changer les idées, autour de bières ou de ti'punchs, en partageant un repas ou avec des discussions pas toujours sensées (ça n'est pas toujours facile d'éviter les conversations de geeks). Je remercie donc tous ceux qui ont partagé avec moi ces moments de détente : les Gros Lourds (Romanus, Rahan, l'Ancêtre, Mémé, ...), Émilie, Estelle, Lolo (tu m'avais bien dit que ça n'allait pas être facile), Max, Seb, Pedro, Poulet & Poulette, Thomas (merci pour la motivation pendant la rédaction), Antoine, Yiannis (bientôt ?), les colocs (Marc & Yak, au boulot et bon courage !), ... Il y en a plein d'autres, des amis de Grenoble, de Guadeloupe ou de Poitiers.

Je remercie également toute ma famille, notamment pour la relecture attentive de ce document. Enfin, je remercie Ния qui m'a supporté et encouragé pendant la rédaction.

Table des matières

Table des figures

Liste des tableaux

Introduction 1

Sommaire

1.1 Problématique

Le calcul haute performance connait un essor important et se démocratise à travers de nombreux domaines scientifiques. Des disciplines comme la physique, la biologie ou la météorologie réalisent des simulations numériques de plus en plus complexes qui nécessitent une puissance de calcul considérable.

Une tendance actuelle consiste à mutualiser des ordinateurs géographiquement éloignés en les interconnectant par un réseau de communication longue distance, par exemple Internet. Cet ensemble de machines ainsi regroupé est appelé une grille de calcul et permet d'offrir à ses utilisateurs une capacité de calcul importante. Cette complexité fait des grilles de calcul un environnement complexe à la fois hétérogène et dynamique.

L'hétérogénéité caractérise la variabilité spatiale du contexte d'exécution, c'est-à-dire que ce contexte diffère selon l'ensemble des ressources utilisées. Les grilles sont composées de machines avec des processeurs et des systèmes d'exploitation variés. La vitesse des processeurs peut varier considérablement entre deux machines. Cette hétérogénéité apparait également au niveau des réseaux de communication : les machines placées sur un même site bénéficient généralement d'un réseau rapide avec une faible latence et un débit important, tandis que les machines géographiquement éloignées communiquent par un réseau longue distance moins performant.

La dynamicité correspond à la variabilité temporelle du contexte d'exécution, c'est-à-dire que l'environnement d'exécution change au cours du temps. Le nombre de ressources disponibles change constamment car elle dépend des réservations et de procédures de maintenance. De plus, la charge des processeurs et des réseaux de communication varie également en fonction de l'activité des autres utilisateurs. Enfin, cette dynamicité peut aussi avoir pour cause des défaillances qui peuvent entrainer la perte, même temporaire, d'un grand nombre de machines.

De part le grand nombre de composants qui constituent une grille de calcul, la probabilité d'une défaillance est importante. Les défaillances sont un cas particulier de variation dynamique du contexte d'exécution et elles nécessitent une prise en compte

particulière. En effet, lors d'une défaillance, une partie de l'état de l'application disparait avec les machines défaillantes. L'application est alors dans un état incohérent qui l'empêche de poursuivre son exécution correctement.

Dans de telles conditions, l'exploitation des ressources d'une grille de calcul est un problème difficile. Pour s'exécuter efficacement et correctement, une application doit nécessairement supporter les contraintes liées à cet environnement d'exécution. Elle doit donc être capable à la fois (1) de tolérer la défaillance d'un ou plusieurs composants et également (2) d'être reconfigurée dynamiquement pour s'adapter aux changements du contexte d'exécution.

1.2 Objectifs

Ces travaux de recherche portent sur l'étude et la conception de mécanismes de reconfiguration dynamique et de tolérance aux fautes pour les applications de calcul haute performance sur des plateformes distribuées de type grille de calcul.

Ces travaux s'intègrent dans le développement du moteur d'exécution Kaapi développé par l'équipe MOAIS au Laboratoire d'Informatique de Grenoble (LIG). L'intergiciel Kaapi permet l'exécution d'applications parallèles et distribuées écrites à l'aide de l'interface de programmation Athapascan. Cette interface permet de décrire le parallélisme d'une application de manière indépendante de l'architecture sur laquelle elle s'exécute. Dans Kaapi, l'application est représentée sous la forme abstraite d'un graphe de flot de données. Grâce à cette représentation abstraite, il est alors possible d'inspecter et de manipuler l'état de l'application en cours d'exécution.

Ces travaux cherchent à fournir des mécanismes de reconfiguration dynamique et de tolérance aux fautes pour le modèle graphe de flot de données offert par Athapascan/Kaapi. Ainsi, l'objectif est de permettre l'exécution d'applications distribuées Kaapi sur une grille de calcul tout en prenant en compte les variabilités de la plateforme et l'apparition de défaillances.

Plus précisément, du point de vue de l'adaptation et de la reconfiguration dynamique, les objectifs sont :

- d'étudier les mécanismes d'adaptation et de reconfiguration dynamique pour le calcul haute performance ;
- de proposer un mécanisme simple, générique et performant pour permettre la réalisation de reconfigurations ;
- d'évaluer de ce mécanisme à grande échelle dans le cadre du moteur d'exécution Kaapi.

Sur l'aspect de la tolérance aux fautes, les objectifs sont :

- d'étudier des mécanismes de tolérance aux fautes pour les architectures distribuées de type grille de calcul ;
- d'intégrer un mécanisme classique de tolérance aux fautes par sauvegarde coordonnée dans l'environnement Kaapi ;
- de proposer et d'étudier les améliorations de cette technique en prenant en compte le modèle de graphe de flot de données.

1.3 Contributions

Les contributions de ce travail de recherche s'articulent autour des problématiques de la reconfiguration dynamique et la tolérance aux fautes.

Une première contribution porte sur la conception d'un **mécanisme de reconfiguration dynamique** pour les applications distribuées. Ce mécanisme est basé sur la représentation de l'application sous forme d'un graphe de flot de données et permet de garantir que l'application restera dans un état correct après la reconfiguration. Il propose la gestion des accès concurrents par **exécution concurrente** ou par **exécution coopérative**. De plus, nous définissons les propriétés de **cohérence** et d'**accessibilité** qui permettent d'assurer la cohérence mutuelle entre les processus de l'application.

Le principe d'exécution coopérative a été utilisé pour réaliser l'implémentation d'une reconfiguration « vol de travail » dans le logiciel X-Kaapi. Il offre de très bonnes performances à grain fin en comparaison de Cilk et de TBB.

Un protocole de gestion de la cohérence mutuelle a été implémenté dans Kaapi. Ce protocole est optimisé pour tenir compte des informations offertes par le graphe de flot de données de l'application. Il a été testé sur un millier de machines de Grid'5000 : il offre à la fois des temps de coordination rapides et stables.

Du point de vue de la tolérance aux fautes, nous avons proposé plusieurs améliorations de la technique classique de sauvegarde/reprise coordonnée. Les protocoles de tolérance aux fautes sont vus comme des reconfigurations et ils sont implémentés à l'aide du mécanisme de reconfiguration proposé précédemment.

Nous avons étudié et amélioré la technique de **reprise globale** qui est associée au protocole de sauvegarde coordonnée. En cas de défaillance, cette technique nécessite classiquement l'utilisation de machines de rechange à la reprise pour ne pas pénaliser la vitesse d'exécution. Afin de supprimer le besoin de machines de rechange, nous proposons de réaliser une **sur-décomposition du domaine de calcul** et d'utiliser un algorithme d'**équilibrage de charge** lors de la reprise. De cette manière, l'application peut redémarrer sur l'ensemble des machines non défaillantes avec des performances proches de l'optimal.

Nous proposons également une méthode originale de **reprise partielle**. Grâce au graphe de flot de données de l'application, elle ne réexécute que le **sous-ensemble de tâches strictement nécessaires à la reprise**. Cette méthode permet de réduire la quantité de travail à réexécuter à la reprise si on la compare à la méthode classique de reprise globale. De plus, si le travail perdu contient suffisamment de parallélisme, il est possible de réexécuter le travail perdu plus rapidement et ainsi de réduire le surcout induit par une panne.

L'étude des performances de ces deux techniques de reprise a été réalisée par des simulations ainsi que par des expérimentations avec une application réelle sur la plateforme Grid'5000. Cela nous a permis de mettre en évidence les gains des améliorations proposées, mais aussi leurs limites.

De manière transversale, une contribution de ces travaux a été l'intégration dans le logiciel Kaapi des mécanismes de reconfiguration dynamique et des protocoles de tolérance aux fautes proposés, mais aussi le développement d'outils pour le **déploiement**

d'application et l'**expérimentation à grande échelle**. Ces outils ont été à la base des victoires de l'équipe Kaapi/MOAIS lors des challenges organisés par l'ETSI dans le cadre des PlugTests 2007 (Pékin, Chine) et 2008 (Sophia-Antipolis).

Les travaux de cette thèse ont fait l'objet de trois publications pour des conférences internationales [30, 29, 78], d'un chapitre de livre [28], d'un article dans un journal national [33], de deux publications pour des conférences nationales [32, 31] et de deux communications orales [27, 26].

1.4 Organisation du manuscrit

Ce document est organisé en trois parties.

La partie I présente l'état de l'art des domaines de la tolérance aux fautes (chapitre 2) et de l'adaptation dynamique (chapitre 3).

Le chapitre 2 introduit les concepts et la terminologie liés à la sureté de fonctionnement. Puis, il présente la tolérance aux fautes pour les systèmes distribués et détaille les techniques basées sur une mémoire stable qui sont plus adaptées au calcul haute performance. Enfin, un aperçu des implémentations réalisées dans la communauté internationale est donné.

Le chapitre 3 explique, tout d'abord, le concept d'adaption dynamique. En particulier, il met en avant les trois fonctions essentielles que sont l'observation, la décision et la reconfiguration. Ensuite, la reconfiguration dynamique pour les applications distribuées est détaillée ; nous y présentons notamment les propriétés qui peuvent être attendues par un mécanisme de reconfiguration. Finalement, nous donnons une description du fonctionnement des principales solutions logicielles qui offrent des fonctions d'adaptation ou de reconfiguration dynamique.

La partie II propose le mécanisme de reconfiguration conçu pour l'environnement Kaapi. Elle est organisée en trois chapitres.

Le chapitre 4 est une présentation du modèle de programmation Athapascan et du moteur d'exécution Kaapi, et notamment du modèle de graphe de flot de données. C'est sur ce modèle qu'est basé le mécanisme de reconfiguration proposé dans le chapitre suivant.

Dans le chapitre 5, nous présentons les mécanismes proposés pour l'environnement Kaapi. Tout d'abord, nous modélisons le processus de reconfiguration. Puis nous détaillons deux aspects fondamentaux qui permettent d'appliquer une reconfiguration tout en garantissant un état correct de l'application : la gestion des accès concurrents et la gestion de la cohérence.

Le chapitre 6 présente les expérimentations qui ont été réalisées pour évaluer les mécanismes de reconfiguration du chapitre précédent. D'abord, nous étudions la gestion de la concurrence à travers les méthodes d'exécution concurrente et coopérative pour la reconfiguration « vol de travail ». Ensuite, le protocole de gestion de la cohérence mutuelle est expérimenté sur une grille de calcul afin de mesurer son surcout.

La partie III présente, à travers trois chapitres, les travaux de cette thèse portant sur la tolérance aux fautes. Ces travaux reposent en partie sur le mécanisme de reconfiguration dynamique présenté dans la partie précédente.

Le chapitre 7 présente tout d'abord l'organisation des mécanismes de tolérance aux fautes dans KAAPI et les cinq composants identifés : le détecteur de panne, le coordinateur des pannes, la mémoire stable, le protocole de sauvegarde et le protocole de reprise. La suite de ce chapitre s'attache à présenter l'implémentation du protocole classique de sauvegarde coordonnée réalisée dans KAAPI. Cette implémentation est également évaluée par une série d'expériences.

Le chapitre 8 étudie la reprise globale qui est le protocole classique de reprise associé à la sauvegarde coordonnée. Nous présentons son implémentation et nous modélisons la phase de reprise. Ensuite, nous étudions, théoriquement et expérimentalement, l'effet de la sur-décomposition d'une application de décomposition de domaine sur la vitesse d'exécution après la reprise.

Dans le chapitre 9, nous proposons un protocole original de reprise partielle utilisant la sauvegarde coordonnée. Ce procotole est présenté et analysé théoriquement. Ensuite, nous l'avons étudié à travers des simulations et plusieurs expérimentations.

Le chapitre 10 conclut ces travaux en rappelant les principaux résultats. Enfin, suite à ces travaux, nous proposons les perspectives de recherche qui nous semblent importantes.

I

État de l'art

Tolérance aux fautes 2

Sommaire

2.1 Introduction

La **tolérance aux fautes** est l'aptitude d'un système informatique à accomplir sa fonction malgré la présence ou l'occurrence de fautes [13], qu'il s'agisse de dégradations physiques du matériel, de défauts logiciels, d'attaques malveillantes, d'erreurs d'interaction homme-machine. Elle apparait comme un moyen de garantir une sureté de fonctionnement.

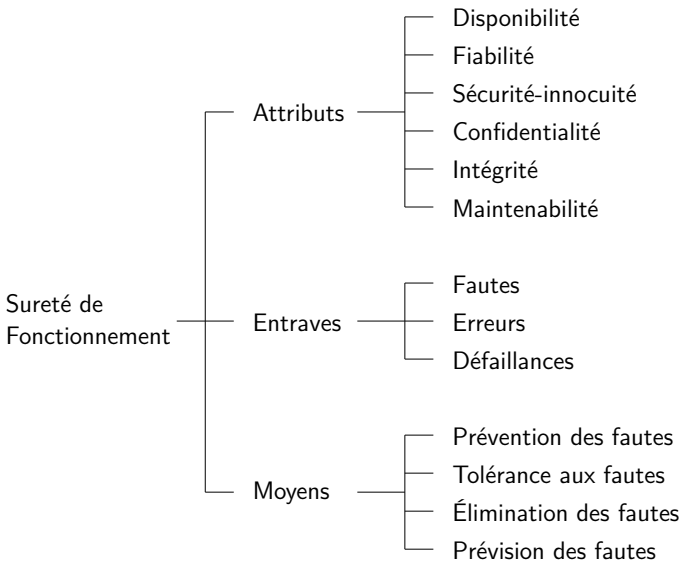


Fig. 2.1: L'arbre de la sureté de fonctionnement [13]

La première section présente les concepts généraux de la sureté de fonctionnement. La seconde section aborde les différentes méthodes de tolérance aux fautes d'une manière très générale. La troisième section donne le modèle d'application distribuée et présente les techniques de tolérance aux fautes spécifiques à ce domaine.

2.2 Sureté de fonctionnement et tolérance aux fautes

La **sureté de fonctionnement** d'un système informatique est son aptitude à délivrer un service en lequel on peut avoir une confiance justifiée [18]. La sureté de fonctionnement peut-être présentée autour de trois notions décrites à la figure 2.1 extraite de [13] : ses **attributs**, ses **entraves** et ses **moyens**.

2.2.1 Attributs de la sureté de fonctionnement

Les attributs de la sureté de fonctionnement correspondent aux propriétés que doit vérifier un système. Ces attributs permettent d'évaluer la qualité de service fournie par le système. Six attributs de la sureté de fonctionnement sont définis dans [18] :

- la **disponibilité** est le fait d'être prêt à l'utilisation ;

- la **fiabilité** correspond à la continuité du service ;
- la **sécurité-innocuité** est l'absence de conséquences catastrophiques engendrées par les fautes ;
- la **confidentialité** correspond à l'absence de divulgation non autorisée de l'information ;
- l'**intégrité** indique l'absence d'altérations inappropriées de l'information ;
- la **maintenabilité** correspond à l'aptitude aux réparations et aux évolutions.

L'importance de chaque attribut peut différer selon l'application et les besoins auxquels le système informatique est destiné. Pour les applications parallèles de longue durée, les principaux attributs seront la fiabilibité et la maintenabilité.

2.2.2 Entraves à la sureté de fonctionnement

Les entraves à la sureté de fonctionnement sont de trois types [13] : les fautes, les erreurs et les défaillances. Fautes, erreurs et défaillances sont liées par des relations de causalité illustrées sur la figure 2.2.

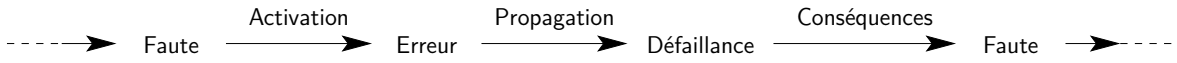


FIG. 2.2: La chaine fondamentale des entraves à la sureté de fonctionnement [13]. Les flèches indiquent les relations de causalité entre fautes, erreurs et défaillances.

Une **défaillance** (ou **panne**) est l'évènement qui survient lorsque le comportement du système dévie de sa fonction[1]. L'**erreur** est la partie de l'état du système qui est susceptible d'entrainer une défaillance. La défaillance survient lorsque l'erreur affecte le service délivré à l'utilisateur. La **faute** est définie comme la cause adjugée ou supposée de l'erreur.

Les fautes sont classées selon huit critères : phase de création ou d'occurence, frontières du système, cause phénoménologique, dimension, intention, capacité et persistance. La combinaison pertinente de ces critères permet de donner un classement exhaustif de tous les types de fautes. Pour simplifier, on peut les regrouper en trois grandes classes non exclusives :

- les fautes de développement qui rassemblent les fautes pouvant survenir durant le développement ;
- les fautes physiques qui rassemblent les fautes affectant le matériel ;
- les fautes d'interactions qui rassemblent les fautes externes, c'est-à-dire celles qui sont localisées à l'extérieur des frontières du système et qui propagent des erreurs à l'intérieur du système par interaction ou interférence.

2.2.3 Moyens d'assurer la sureté de fonctionnement

Les moyens d'assurer la sureté de fonctionnement sont définis comme les méthodes utilisées pour assurer cette propriété. On distingue quatre méthodes principales [18].

[1] et non de la spécification du système puisqu'elle peut être erronée

- La **prévention des fautes** vise à empêcher l'apparition ou l'introduction des fautes dans le système. Elle repose sur des règles de développement (modularisation, utilisation de langage fortement typé, preuve formelle, etc.).
- L'**élimination des fautes** s'attache à réduire la présence (nombre, sévérité) des fautes. Cette méthode opère à la fois lors du développement (vérification des conditions, test de régressions, injection de fautes, etc.) ou lors de l'utilisation (maintenance).
- La **prévision des fautes** cherche à estimer (qualitativement et quantitativement) l'occurrence et les conséquences des fautes. Elle est réalisée par la modélisation et l'évaluation de systèmes.
- La **tolérance aux fautes** essaie de masquer l'occurrence des fautes et de continuer à fournir le service demandé malgré leur apparition.

La section suivante va présenter les différentes approches pour réaliser la tolérance aux fautes.

2.2.4 Tolérance aux fautes

L'objectif de la tolérance aux fautes est d'éviter les défaillances du système malgré la présence de fautes. Cela revient à casser la chaine décrite à la figure 2.2, qui conduit de la faute à la défaillance. La tolérance aux fautes est mise en œuvre par la **détection d'erreur** et le **rétablissement du système**.

2.2.4.1 Détection d'erreur

La détection d'erreur peut être réalisée lors d'une suspension de service. On dit alors qu'elle est préemptive. À l'opposé, on dit qu'elle est concomitante lorsqu'elle est réalisée lors de l'exécution normale du service.

Les techniques de détection concomitante utilisent la redondance au niveau information ou composant, ou la redondance temporelle ou algorithmique. Les formes les plus utilisées sont les suivantes.

- Les codes détecteur d'erreur : ils introduisent une redondance dans la représentation de l'information [150].
- Le doublement et la comparaison : les unités de traitement sont dupliquées et leurs résultats sont comparés.
- Les contrôles temporels et d'exécution : un « chien de garde » (*watchdog*) contrôle les temps de réponse ou l'avancée de l'exécution.
- Les contrôles de vraisemblance ou de données structurées : des assertions sont insérées dans le code pour vérifier des types, des indices, des valeurs, etc.

2.2.4.2 Rétablissement du système

Le rétablissement du système vise à transformer l'état erroné en un état exempt d'erreur et de faute. Le traitement de la faute se fait en identifiant le composant fautif et en l'excluant. Le traitement de l'erreur peut se faire par trois techniques : la reprise, la poursuite et la compensation [13].

La **reprise** est la technique la plus couramment utilisée. L'état du système est sauvegardé régulièrement. Lorsqu'une erreur est détectée, le système est ramené à

un état antérieur à l'occurrence de l'erreur. Cet état sauvegardé est appelé **point de reprise**.

La **poursuite** consiste à rechercher un nouvel état exempt d'erreur. Ceci peut par exemple être réalisé en associant un traitement exceptionnel lorsqu'une erreur est détectée. Le but de ce traitement est alors de corriger l'état erroné.

La **compensation** nécessite que l'état du système comporte suffisamment de redondance pour permettre sa transformation en un état exempt d'erreur. Elle est transparente vis-à-vis de l'application car elle ne nécessite pas de réexécuter une partie de l'application (reprise), ni d'exécuter une procédure dédiée (poursuite). Elle peut par exemple être réalisée en répliquant des composants et en effectuant un vote majoritaire sur les résultats. Une autre manière de procéder est d'utiliser les codes correcteurs d'erreurs [150] ou plus généralement des algorithmes tolérants aux fautes [88, 38, 152, 131].

On peut noter que la méthode de compensation ne nécessite pas de détection d'erreurs spécifique puisqu'elle effectue elle-même la détection d'erreur. Une méthode de compensation peut servir de détecteur d'erreur, tandis que l'inverse n'est pas vrai. En effet, la compensation nécessite une redondance plus importante pour pouvoir corriger l'erreur [13]. Par exemple en termes de composants, deux composants suffisent à détecter une erreur, mais au moins trois seront nécessaires pour la corriger.

Dans la suite de ce chapitre, nous nous intéressons à la tolérance aux fautes pour les systèmes distribués, et en particulier aux méthodes de rétablissement du système.

2.3 Tolérance aux fautes pour les systèmes distribués

Un système distribué est constitué par l'aggrégation d'un très grand nombre de composants. Une faute d'un seul des composants entraine la défaillance de tout le système. Ainsi, même si chaque composant présente une probabilité de faute très faible, la défaillance du système est inévitable [142].

La tolérance aux fautes apparait comme un élément indispensable aux systèmes répartis. Pour répondre à ce besoin, plusieurs techniques ont été conçues. Elles reposent toutes sur un mécanisme de redondance [16]. Dans un premier temps, nous présentons d'abord une modélisation du système distribué puis nous décrivons les mécanismes de redondance.

2.3.1 Modèle et hypothèses

Un **système distribué** est modélisé comme un ensemble de processus qui communiquent en échangeant des messages par l'intermédiaire de canaux de communication. L'état du système est donc distribué sur l'ensemble des processus et il n'y a pas d'horloge globale.

Pour caractériser les canaux de communication, plusieurs modèles temporels existent.

- Le modèle synchrone indique que la durée de transfert (*i.e.* le temps entre l'émission et la réception) des messages est bornée. Cependant ce modèle n'est pas suffisamment proche de la réalité puisqu'il nécessite que la borne de temps soit toujours respectée.

- Le modèle asynchrone garantit qu'un message émis sur le canal de communication sera délivré au destinataire, cependant il n'est pas possible de borner la durée de transfert du message. En présence de défaillances, ce modèle ne permet pas de résoudre certains problèmes fondamentaux comme le consensus [73].
- Le modèle asynchrone avec détecteur de défaillance [48] permet de résoudre le problème du consensus en présence de défaillances tout en conservant des hypothèses réalistes vis-à-vis des communications.

Par la suite, nous nous plaçons dans un modèle asynchrone avec détecteur de défaillance (ou dans un modèle équivalent). De plus, les canaux de communication sont supposés fiables et ordonnés (*First In, First Out*)[2].

Modèle de pannes. Les défaillances (ou pannes) potentielles d'un service peuvent prendre des formes variées. Les classements proposés pour distinguer les différents types de défaillances s'appuient sur différents critères. Les principaux critères utilisés dans la littérature [17, 92] pour caractériser les défaillances des processus ou des canaux de communication sont les suivants.

- Selon la gravité, on distingue alors les arrêts, les omissions (des messages sont perdus), les défaillances de temporisation (le temps de réaction du système est en dehors des plages spécifiées), les défaillances en valeurs (les résultats fournis sont incorrects) et les défaillances incohérentes ou byzantines.
- Selon la persistance temporelle, on trouve les défaillances transitoires (isolées dans le temps), les défaillances intermittentes (aléatoires et répétées) et les défaillances permanentes (définitives jusqu'à réparation).
- Selon l'intention de la faute, on différencie les défaillances malveillantes et les défaillances non malveillantes.

2.3.2 Gestion de la redondance

La tolérance aux fautes dans un système distribué est assurée par la redondance. Cette redondance peut être spatiale (réplication de composants), temporelle (traitements multiples) ou informationnelle (redondance de données, codes, signatures). Les mécanismes de redondances mis en œuvre appartiennent à deux catégories : les mécanismes utilisant la réplication et les mécanismes s'appuyant sur une mémoire stable. Ces mécanismes sont présentés dans les paragraphes suivants.

2.3.2.1 Redondance spatiale et temporelle : réplication

La tolérance aux fautes par réplication consiste à utiliser des copies multiples d'un même composant ou processus. De cette manière, en cas de défaillance d'un des composants, la défaillance peut être masquée par l'une des copies. La principale difficulté de cette approche est de conserver une cohérence forte entre les copies. Il existe quatre stratégies principales permettant d'assurer cette cohérence [63] :

- Pour la **réplication passive** [155], on distingue la copie primaire et les copies secondaires. La copie primaire est la seule qui reçoit les requêtes et qui effectue toutes les opérations. Pour assurer la cohérence, la copie primaire diffuse son

[2] Ces hypothèses peuvent être garanties par un protocole de communication sous-jacent.

nouvel état aux copies secondaires après chaque modification. Cet état sert de point de reprise en cas de défaillance.

- La **réplication active** [155] désigne les stratégies dans lesquelles toutes les copies jouent un rôle identique. Toutes les copies reçoivent la même séquence ordonnée de requêtes, qui sont toutes traitées dans le même ordre. Cette stratégie évite d'utiliser des points de reprise couteux. En revanche, elle nécessite un mécanisme de diffusion atomique et requiert que l'exécution des requêtes soit déterministe pour garantir la cohérence.
- La **réplication semi-active** [63] est une amélioration de la réplication active. À la différence de la réplication active, les copies secondaires attendent une notification de la copie primaire avant de traiter la requête. Cette notification comporte les informations nécessaires qui permettent de résoudre le problème de l'indéterminisme du traitement des requêtes.
- La **réplication coordinateur/cohortes** [63] est également une solution hybride entre la réplication active et la réplication passive. La copie primaire est appelée coordinateur et les copies secondaires sont appelées cohortes. Cette méthode est une réplication passive pour laquelle les requêtes sont transférées à toutes les copies pour éviter de les perdre en cas de défaillance.

Le principal désavantage de cette méthode par réplication est qu'elle nécessite de nombreuses ressources : pour tolérer p défaillances, il est nécessaire d'avoir $p + 1$ composants identiques. Cette méthode n'est donc pas adaptée aux calculs parallèles où la performance (temps de calcul) est souvent le critère prépondérant : les ressources doivent être exploitées en priorité pour le calcul.

2.3.2.2 Redondance informationnelle : mémoire stable

La mémoire stable représente un support de stockage. Son rôle est de conserver les sauvegardes des informations du système qui permettront de reprendre l'exécution de l'application dans un état cohérent [66]. Une mémoire stable doit préserver l'intégrité des données et les garder accessibles, même en cas de défaillance.

La réalisation physique d'une mémoire stable dépend essentiellement des types de défaillances auxquels on souhaite faire face.

- Pour un système qui ne tolère qu'une seule défaillance (respectivement p défaillances), la mémoire stable peut être réalisée par la mémoire volatile d'un autre processus (respectivement de p autres processus).
- Dans un système qui ne souhaite tolérer que les défaillances transitoires, la mémoire stable peut correspondre au disque dur local du processus.
- Pour un système qui veut tolérer un nombre quelconque de défaillances permanentes, la mémoire stable doit être réalisée sur une machine extérieure aux machines de calcul et qui est supposée être fiable.

Le principe de la tolérance aux fautes par mémoire stable est, en cas de défaillance, de rétablir l'application dans un état cohérent en utilisant les informations stockées sur la mémoire stable. Dans la suite de chapitre, nous étudierons uniquement les protocoles basés sur une mémoire stable.

2.4 Techniques de tolérance aux fautes par mémoire stable

Le début de ce chapitre a présenté les concepts sous-jacents à la réalisation d'un système distribué tolérant aux fautes. Nous présentons ici les techniques de tolérance aux fautes basées sur l'utilisation d'une mémoire stable car elles sont plus adaptées au calcul parallèle haute performance. Dans le cadre de ce mémoire, nous considérons uniquement cette approche.

La tolérance aux fautes basée sur une mémoire stable utilise la redondance d'informations. Ces informations correspondent à la sauvegarde de l'état des processus ou bien à la journalisation d'évènements, et sont stockées sur la mémoire stable.

Une difficulté majeure de cette approche est la constitution, à partir des informations sauvegardées, d'un état correct du système prenant en compte tous les processus. La constitution d'un tel état introduit un surcout qui va dépendre des contraintes imposées au système : nombre et types de défaillances à tolérer, reprise globale du système ou uniquement des processus défaillants, etc. La conception du protocole de tolérance aux fautes doit prendre en compte ces contraintes tout en limitant la dégradation de performance infligée au système.

Deux approches sont possibles pour construire un état global cohérent suivant le moment de sa construction [66, 49].

À priori : les sauvegardes des processus sont coordonnées à l'exécution pour constituer un état global cohérent.

À posteriori : les sauvegardes sont faites de manière indépendante et l'état global cohérent est construit à la reprise.

Si la première approche garantit un état correct par construction de la sauvegarde, la seconde approche peut ne pas arriver à reconstruire cet état correct sans hypothèse supplémentaire.

Voyons maintenant plus précisément comment définir un état correct ou cohérent.

2.4.1 État global cohérent

L'état global d'une application parallèle est composé :
- de l'état local de tous les processus participant au calcul,
- et de l'état de tous les canaux de communication entre les processus.

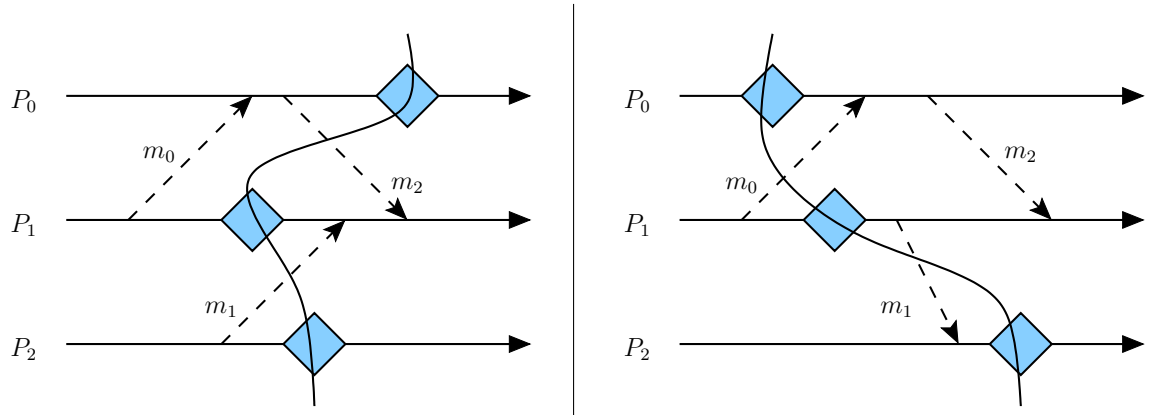
Sans horloge globale, il n'est pas possible d'observer de manière simultanée l'état des processus. De plus l'état des canaux de communication n'est pas accessible directement. Ces contraintes empêchent de connaitre à un instant donné l'état global de l'application, et donc de le sauvegarder.

En pratique, l'état global de l'application est donc reconstitué à partir des états des processus observés à des instants différents et des informations sur les canaux de communication acquises sur les processus émetteurs et récepteurs. Chandy et Lamport ont donc défini la notion d'état global cohérent de la manière suivante.

Définition 1 *Un **état global cohérent** est un état qui peut se produire durant une exécution correcte (*i.e. *sans défaillance) de l'application. Plus formellement, un état*

global cohérent est un état dans lequel, si l'état d'un processus montre la réception d'un message, alors l'état du processus qui a émis ce message « contient[3] *» l'émission de ce message [66, 49].*

Dans la figure 2.3b, l'état global, formé des losanges sur chacune des lignes de temps des trois processus P_0, P_1 et P_2 et des canaux de communication, est incohérent car le message m_1 est enregistré comme reçu dans l'état sauvegardé du processus P_2 alors que la sauvegarde du processus P_1 ne montre pas l'émission du message m_1. On peut remarquer que ce message ne fait pas partie de l'état des canaux de communication car il n'existe pas encore lorsque P_1 sauvegarde son état local et il n'existe plus lorsque P_2 sauvegarde son état local. Ce message m_1 est appelé « message orphelin ».



(a) État global cohérent (b) État global incohérent

FIG. 2.3: Exemple d'état global cohérent (a) et incohérent (b). Les axes horizontaux représentent le temps pour trois processus P_0, P_1 et P_2. Les flèches représentent des communications entre les processus. Les losanges représentent les évènements associés à la capture de l'état des processus.

Définition 2 *Soit un état global, on appelle* ***message orphelin*** *est un message qui apparait comme reçu dans l'état d'un processus alors qu'il n'apparait pas dans l'état du processus qui l'a émis.*

Les messages orphelins sont la cause de l'incohérence d'un état global car ils correspondent à un état qui ne peut pas se produire lors d'une exécution correcte. Ils proviennent du décalage des sauvegardes locales qui peut se produire entre deux processus distants qui communiquent.

Le rôle du protocole de tolérance aux fautes est de reconstruire un état global cohérent à partir de l'état potentiellement incohérent du système après une panne et des informations sauvegardées sur la mémoire stable. L'état global cohérent reconstruit n'est pas nécessairement un état de l'application avant la défaillance ; il suffit qu'il soit un état de l'application qui aurait pu se produire durant une exécution sans panne.

La section suivante présente les protocoles de reprise basés sur la sauvegarde. Puis nous présenterons les protocoles basés sur la journalisation des évènements non déterministes.

[3] signifie que l'évènement associé à l'émission précède l'évènement associé à la prise d'état du processus.

2.4.2 Reprise par sauvegarde

Les protocoles de reprise par sauvegarde réalisent des sauvegardes régulières de l'état des processus. Pour redémarrer, ils utilisent le dernier ensemble de sauvegardes formant un état global cohérent [66].

Les protocoles de reprise par sauvegarde peuvent être classés en trois catégories selon le mode de construction de l'état global cohérent de la reprise : la sauvegarde coordonnée, la sauvegarde non coordonnée et la sauvegarde induite par les communications.

2.4.2.1 Sauvegarde non coordonnée

La sauvegarde non coordonnée [66] évite la coordination et laisse à chaque processus la décision de sauvegarder son état quand il le souhaite. Ainsi un processus peut décider de sauvegarder son état quand ça lui convient le mieux, par exemple quand la taille de son état est minimale [151]. Lors de la reprise, un algorithme analyse les différences entre les points de sauvegarde des processus pour tenter de déterminer l'ensemble des sauvegardes les plus récentes formant un état global cohérent.

Cependant cette approche comporte plusieurs inconvénients. Tout d'abord, lors de la reprise, il y a un risque d'**effet domino** [123] : lors de la construction de l'état global cohérent, les dépendances entre les messages peuvent entrainer un retour à l'état initial. On peut ainsi perdre une grande partie du travail déjà effectué. De plus, certaines sauvegardes peuvent être inutiles pour la construction d'un état global cohérent. Ces sauvegardes induisent un surcout mais ne contribuent pas au redémarrage. Enfin, cette méthode oblige les processus à conserver à priori toutes les sauvegardes. Un ramasse-miette peut être utilisé pour supprimer les sauvegardes inutiles en détectant l'ensemble des sauvegardes le plus récent qui constitue un état global cohérent.

2.4.2.2 Sauvegarde coordonnée

La sauvegarde coordonnée est réalisée en coordonnant les processus de manière à assurer que l'ensemble des états des processus forme un état global cohérent. Il existe différentes manières de coordonner les processus. On distingue en particulier la sauvegarde coordonnée bloquante ou la sauvegarde coordonnée non bloquante.

La **sauvegarde coordonnée bloquante** [138, 118] est réalisée en plusieurs étapes. Tout d'abord, tous les processus de calcul sont arrêtés et les canaux de communication sont vidés. Puis chaque processus sauvegarde son état local. Enfin, les calculs peuvent reprendre. Les canaux de communication étant vides au moment de la sauvegarde, l'état global de l'application correspond à l'état local de tous ses processus et il est cohérent.

La **sauvegarde coordonnée non bloquante** [49] permet d'identifer l'état des canaux de communication grâce à des « messages marqueurs ». Lors d'une étape de sauvegarde, chaque processus sauvegarde son état local puis diffuse immédiatement[4] un marqueur sur tous ses canaux de communication. Ensuite, il sauvegarde tous les messages reçus sur chaque canal de communication jusqu'à la réception d'un marqueur. Cet ensemble de messages correspond à l'état du canal de communication qui sera alors sauvegardé sur la mémoire stable.

[4] avant tout autre message

L'avantage de la sauvegarde coordonnée est qu'elle n'est pas sensible à l'**effet domino** lors de la reprise. Seule la dernière sauvegarde est nécessaire pour un redémarrage ce qui réduit le surcout de stockage. Le principal inconvénient est le surcout induit par la synchronisation des processus. D'autres méthodes ont été proposées pour tenter d'améliorer les performances :

- La sauvegarde avec **horloges synchronisées** [100] synchronise les horloges des processus. Si chaque processus effectue sa sauvegarde et s'il attend un temps suffisant (dépendant des déviations entre les horloges et du temps de détection d'une défaillance), on est assuré de la cohérence des sauvegardes sans avoir échangé de messages.
- La sauvegarde coordonnée **minimale** [96] ne synchronise un processus qu'avec les processus dont il dépend réellement. Ceci est réalisé en deux étapes. Durant la première, un processus identifie les processus dont il dépend (il émet des messages qui sont diffusés de proche en proche selon les dépendances entre processus) ; durant la deuxième, les processus identifiés réalisent leur sauvegarde.

2.4.2.3 Sauvegarde induite par les communications

La sauvegarde induite par les communications (*Communication-Induced Checkpointing* ou CIC) [19, 9] est un compromis entre la sauvegarde coordonnée et la sauvegarde non coordonnée. Ce protocole utilise deux types de points de sauvegarde : les sauvegardes locales et les sauvegardes forcées. Les sauvegardes locales sont des sauvegardes que le processus décide d'effectuer indépendamment. Les sauvegardes forcées sont des sauvegardes qui doivent être effectuées pour empêcher l'effet domino en cas de reprise.

Ce protocole repose sur les notions de Z-chemin (zigzag path) et de Z-cycle définies dans [113]. Ces notions permettent de déterminer si une sauvegarde sera utile ou non en cas de panne et de reprise.

La sauvegarde induite par les communications a été spécialisée pour l'ordonnancement par vol de travail. Cette variante, appelée TIC (*Theft-Induced Checkpointing*) [92, 94, 93] considère que dans le cas du vol de travail, les seules communications qui modifient l'état de l'application sont les requêtes et les retours de vol.

2.4.3 Reprise par journalisation

Le principe de la tolérance aux fautes par journalisation est de sauvegarder l'histoire de l'application. Les protocoles par journalisation utilisent à la fois la sauvegarde locale de l'état des processus et la journalisation des évènements déterministes pour permettre la reprise de l'application. Il est alors possible de reprendre l'exécution des processus défaillants (et uniquement des processus défaillants) à partir de leur dernière sauvegarde en rejouant les évènements non déterministes sauvegardés.

Pour cela, les protocoles basés sur la journalisation reposent sur l'hypothèse PWD :

Définition 3 ***Hypothèse PWD*** *(*PieceWise Deterministic assumption*) [137]*

- *Un processus est modélisé par une séquence d'intervalles d'état.*
- *Chaque intervalle débute par un évènement non déterministe.*
- *L'exécution du processus durant chaque intervalle est déterministe.*

Chaque processus crée un journal des évènements non déterministes qu'il observe pendant l'exécution normale (sans panne). En cas de panne, le processus défaillant est capable de reconstruire son état d'avant la panne en partant de sa dernière sauvegarde et en rejouant les évènements non déterministes inscrits dans son journal. L'hypothèse PWD garantit que cette reconstruction aboutira au même état.

Pour pouvoir bénéficier de cette hypothèse, il faut être capable de détecter et d'enregistrer ces évènements non déterministes. Ces évènements peuvent être des réceptions de messages ou des évènements internes au processus (des décisions d'ordonnancement par exemple). Il faut également remarquer que ces informations (le journal et les sauvegardes périodiques) doivent être stockées sur une mémoire stable.

Lors de la reprise par journalisation, seuls les processus défaillants retournent en arrière. La reprise force l'exécution des processus redémarrés à être identique à celle qui s'est produite avant la panne. L'état obtenu après la reprise est donc exactement l'état de l'application d'avant la défaillance. Cet état global est nécessairement cohérent puisqu'il correspond à un état de l'application lors d'une exécution sans panne.

La reprise par journalisation n'est donc pas sensible à l'effet domino. C'est pourquoi la journalisation est souvent utilisée conjointement à la sauvegarde non coordonnée.

Sous l'hypothèse PWD, il est possible de définir la notion de processus orphelin et la condition de « non-orphelinité ».

Définition 4 *On appelle* ***processus orphelin*** *un processus dont l'état dépend d'un évènement non déterministe qui ne peut être reproduit à la reprise [66].*

L'existence de ces processus orphelins caractérise les états globaux « pathologiques » pour lesquels il n'est pas possible de reprendre une exécution. En effet, un processus orphelin est un processus dont l'état dépend d'un évènement non déterministe qui ne pourra pas être reproduit à la reprise.

Pour pouvoir redémarrer l'application, les protocoles de journalisation doivent assurer la condition de « non-orphelinité ».

Proposition 1 ***Condition de « non-orphelinité »*** *Lors de la reprise des processus défaillants, le système ne doit contenir aucun processus orphelin.*

Une formalisation de cette condition de « non-orphelinité » peut être trouvée dans [66].

Les protocoles de reprise par journalisation diffèrent par leur manière d'assurer et d'implanter cette condition de « non-orphelinité ». La suite présente la journalisation pessimiste, la journalisation optimiste et la journalisation causale.

2.4.3.1 Journalisation pessimiste

Le protocole de reprise par journalisation pessimiste [137] repose sur l'hypothèse (pessimiste) qu'une défaillance peut se produire immédiatement après un évènement non déterministe. Le principe de ce protocole est donc de ne pas permettre à un processus de dépendre d'un évènement non déterministe tant que celui-ci n'a pas été stocké sur un support stable.

Concrètement, si on considère que les évènements non déterministes sont uniquement les réceptions de messages, ce protocole impose aux processus de sauvegarder tous les messages reçus avant d'émettre un message à un autre processus [8]. Les sauvegardes effectuées doivent donc être réalisées de manière synchrone.

L'avantage de ce protocole est qu'il ne crée jamais de processus orphelin. Cependant, la sauvegarde synchrone induit un surcout conséquent lors d'une exécution sans panne, en particulier pour les applications effectuant beaucoup de communications.

2.4.3.2 Journalisation optimiste

La journalisation optimiste [137] veut améliorer les performances en faisant l'hypothèse (optimiste) qu'une panne ne se produira pas avant la sauvegarde de l'évènement non déterministe. Ainsi, la contrainte est relâchée et les sauvegardes peuvent être réalisées de manière asynchrone.

Cependant, le désavantage de cette méthode est qu'elle ne garantit pas strictement la « condition de non-orphelinité ». Ainsi les processus qui n'ont pas encore sauvegardé leurs évènements non déterministes sont des processus orphelins. Pour obtenir un état global cohérent, ces processus devront revenir en arrière au dernier état où ils n'étaient pas orphelins. Il faut remarquer que ce calcul pour obtenir l'état global cohérent à la reprise peut avoir un cout important.

2.4.3.3 Journalisation causale

La journalisation causale [65, 66] combine les avantages de la journalisation pessimiste pour la reprise et les avantages de la journalisation optimiste en ce qui concerne le surcout à l'exécution. L'inconvénient est sa complexité.

Le principe est de conserver en mémoire locale les informations permettant de rejouer un évènement non déterministe mais également les informations de précédence (au sens de la relation de précédence causale de Lamport [101]) avec les autres évènements non déterministes [8]. Ces informations (appelées également le déterminant) sont aussi ajoutées à tous les messages émis vers les autres processus. Ces informations sont retirées de la mémoire locale des processus une fois qu'elles ont été enregistrées sur un support stable.

Ainsi, à tout moment, un processus connait l'historique des évènements qui ont produit son état et celui des autres processus. Ces informations le protègent des défaillances des autres processus et permettent de garantir la condition de « non-orphelinité » [65].

2.4.4 Comparaison des protocoles

Le tableau 2.1 propose une comparaison des avantages et des inconvénients de différentes techniques de tolérance aux fautes par reprise. Les critères utilisés sont les suivants.

Hypothèse PWD : indique si cette technique repose sur l'hypothèse PWD. On remarque que seuls les protocoles par journalisation font cette hypothèse.

Processus orphelins : indique si l'état correspondant à la dernière sauvegarde peut contenir des processus orphelins. Les processus orphelins peuvent être évités en utilisant plusieurs sauvegardes.

Effet domino : indique s'il y a un risque d'effet domino au moment de la reprise. L'effet domino va obliger à conserver toutes les sauvegardes pour perdre le moins de calculs possibles. Les techniques par journalisation ne sont pas sensibles à l'effet domino puisqu'elles ont sauvegardé les messages pouvant en être à l'origine.

Nombre de sauvegardes : donne le nombre de sauvegardes par processus à conserver pour redémarrer dans un état global cohérent. Ceci est la conséquence de la possibilité d'apparition de processus orphelins et de l'effet domino.

Protocole	Hypothèse PWD	Processus orphelins	Effet Domino	Nombre de sauvegardes
Journalisation pessimiste	Oui	Non	Non	Une
Journalisation optimiste	Oui	Possible	Non	Plusieurs
Journalisation causale	Oui	Non	Non	Une
Sauvegarde coordonnée	Non	Non	Non	Une
Sauvegarde non coordonnée	Non	Possible	Possible	Toutes
Sauvegarde induite par les communications	Non	Possible	Non	Plusieurs

Tab. 2.1: Comparaison des différentes méthodes de tolérance aux fautes par reprise

2.5 Implémentations existantes

Cette section donne un aperçu et une comparaison des principales implémentations de protocoles de tolérance aux fautes pour les applications parallèles de calcul distribué.

De nombreuses implémentations des protocoles décrits précédemment ont été faites. Cette section va présenter les principales implémentations dans le cadre des environnements de programmation parallèle. Un environnement complet de tolérance aux fautes est un système complexe. Nous allons donc d'abord présenter les implémentations de sauvegarde locale de l'état d'un processus, puis les implémentations de protocoles de tolérance aux fautes proprement dites.

2.5.1 Sauvegarde locale d'un processus

Le problème de la sauvegarde locale de l'état d'un processus peut être abordé de trois manières différentes, dépendant du niveau (dans la pile logiciel) auquel elle est effectuée.

La première approche se situe au niveau système : l'état du processus est sauvegardé comme un espace mémoire. La sauvegarde peut alors être faite par le noyau comme avec Berkeley Lab's Linux Checkpoint/Restart (BLCR) [64], ou bien par une bibliothèque

utilisateur comme Condor [104], Libckpt [120] ou MTCP [129]. En particulier, Libckpt offre un mécanisme de sauvegarde incrémentale et de compression qui permet de réduire le volume de données sauvegardées [122]. Parmi ces solutions, seuls BLCR et MTCP supportent les applications multithreadées. Cette méthode est très utilisée parce qu'elle est transparente pour le développeur de l'application, mais elle comporte plusieurs inconvénients : elle requiert des ressources homogènes pour le redémarrage (même système d'exploitation et même architecture processeur) ; et l'espace mémoire contient des données inutiles au redémarrage, ce qui implique que la taille du checkpoint est plus grande que nécessaire.

Dans le but d'abstraire l'état d'un processus, la deuxième approche considère que la responsabilité de l'utilisateur est d'écrire les fonctions pour sauvegarder et restaurer l'état d'un processus. Cette méthode est efficace car le développeur peut choisir exactement quelles données doivent être sauvegardées, mais cela nécessite un effort supplémentaire de la part du développeur de l'application.

La troisième approche opère au niveau de l'intergiciel. Elle combine les avantages des deux approches précédentes, mais elle requiert que l'application soit écrite avec un intergiciel qui utilise une représentation abstraite de l'application. Cette représentation abstraite peut prendre la forme d'objets (Charm++ [86]), de listes de tâches (Satin [159]) ou d'un graphe de flot de données (Kaapi [93, 92]). Grâce à cette représentation abstraite, l'intergiciel peut sauvegarder lui-même les données représentant l'état de l'application. Cette approche est totalement transparente pour le développeur de l'application ; un processus peut être restauré sur une ressource hétérogène (la représentation abstraite est indépendante de l'architecture) et la taille de la sauvegarde est plus petite que l'espace mémoire du processus.

2.5.2 Implémentations de protocoles de tolérance aux fautes

L'objectif des environnements de calcul de tolérance aux fautes est d'offrir un moyen simple de rendre une application tolérante aux fautes. On distingue en particulier les méthodes semi-automatiques et les méthodes automatiques.

Dans la catégorie semi-automatique, on trouve FT-MPI [71] et LA-MPI [15]. À l'apparition d'une panne, l'environnement de calcul survit et remonte une erreur au niveau de l'application qui peut alors la traiter et réagir de manière adéquate. Les intergiciels semi-automatiques peuvent offrir de bonnes performances car ils permettent de spécialiser la méthode de tolérance aux fautes pour une application donnée, mais ils manquent de transparence pour le développeur de l'application.

Les méthodes automatiques sont beaucoup plus nombreuses. Elles reposent sur les techniques de tolérances aux fautes proposées dans ce chapitre.

Tout d'abord, FT/MPI [20] et P2P-MPI [126] implémentent des techniques de tolérance aux fautes basées sur la réplication. Les processus sont répliqués, et de cette manière, la défaillance d'un processus répliqué n'affectera pas le calcul. Ces méthodes évitent les interruptions de service mais elles visent les platformes avec un très grand nombre de machines puisqu'elles utilisent beaucoup de ressources.

CoCheck [136] est une des premières solutions à offrir de la tolérance aux fautes à MPI (1996). Il utilise un protocole de sauvegarde coordonnée bloquante pour garantir

la cohérence de l'état global de l'application, et Condor [104] pour la sauvegarde locale de chaque processus. La technique de tolérance aux fautes par sauvegarde coordonnée est très répandue. Elle a été implémentée en différentes variantes et optimisée pour MPI avec Starfish MPI [2], LAM/MPI [133], MPICH-V [44, 102, 56], Open MPI [90] et aussi avec d'autres modèles de programmation dans Charm++ [164]. DMTCP [12] propose aussi une sauvegarde coordonnée bloquante tout en étant indépendant du modèle de programmation.

La sauvegarde non coordonnée est utilisée sans journalisation dans Starfish MPI [2]. Starfish MPI utilise des communications de groupe atomiques qui garantissent une diffusion fiable et ordonnée des messages et il n'est donc pas sensible à l'effet domino. Cependant la sauvegarde non coordonnée est principalement utilisée conjointement à un protocole de journalisation pour éviter l'effet domino.

Quant à la sauvegarde induite par les communications, une implémentation spécialisée pour le vol de travail est réalisée dans Kaapi avec le TIC (*Theft-Induced Checkpointing*) [92, 94, 93]. Dans ProActive, un protocole de sauvegarde induite par les communications, étendu par un mécanisme de journalisation des messages, a également été implémenté [21].

Les protocoles par journalisation ont aussi largement été étudiés. L'environnement Egida [124, 125] offre son propre langage pour exprimer les différents protocoles de journalisation : pessimiste, optimiste et causale.

Des implémentations de la journalisation pessimiste sont proposées dans MPI-FT [105], MPICH-V [40, 42] et Charm++ [47]. La journalisation causale a été expérimentée à travers Manetho [65] et MPICH-V [102].

Finalement, Satin [159] offre un service de tolérance aux fautes par une appproche différente des protocoles classiques. Cette approche est brièvement présentée dans la section suivante. Toutes ces implémentations de protocoles de tolérance aux fautes sont rarement comparées entre elles. On peut noter que cet effort a été fait dans les environnements Egida [124, 125] et MPICH-V [43] qui implémentent chacun plusieurs variantes de ces protocoles.

2.5.3 Comparaison des implémentations

Cette section détaille quelques environnements de programmation parallèle récents qui implémentent des mécanismes de tolérance aux fautes. Ils ont été choisis parce qu'ils sont largement utilisés ou parce qu'ils proposent une approche originale du problème de la tolérance aux fautes. Ils représentent un aperçu des solutions automatiques actuelles pour la tolérance aux fautes.

Ces implémentations sont comparées dans le tableau 2.2 en utilisant les critères suivants.

Sauvegarde locale : indique les méthodes utilisées pour sauvegarder localement un processus parmi celles décrites à la section 2.5.1. Ceci influence directement la portabilité et la taille de l'état sauvegardé.

Protocole : correspond au protocole de tolérance aux fautes utilisé. Les protocoles classiques sont détaillés dans la section 2.4.

Composants de stockage : donne le composant physique qui est utilisé pour conserver les états sauvegardés ou les messages enregistrés.

Composants fiables : définit quels composants sont supposés fiables dans cette implémentation.

Processeur de rechange : indique si un processeur de rechange est nécessaire pour redémarrer l'application. Redémarrer sans processeur de rechange requiert un système d'équilibrage de charge pour empêcher une baisse des performances.

Type de reprise : définit combien de processus doivent reprendre à un état antérieur lors du redémarrage : *globale* pour tous, *locale* pour seulement les processus défaillants, et *partielle* pour une solution intermédiaire.

2.5.3.1 MPICH-V

MPICH-V offre une implémentation de MPI pour des ressources volatiles. Il est basé sur l'implémentation MPICH du standard MPI-1. Les protocoles de tolérance aux fautes sont implémentés en remplaçant le composant standard *ch_p4* de MPICH (réalisant les communications TCP) par un composant *V-protocol* qui représente le protocole de tolérance aux fautes. Les versions récentes de MPICH-V peuvent utiliser Condor, Libckpt ou BLCR pour la sauvegarde locale de processus.

Plusieurs protocoles de tolérance aux fautes ont été implémentés dans MPICH-V. MPICH-V1 [40] est une implémentation de la journalisation pessimiste basée sur le concept de canal mémoire (*memory channel*). Un canal mémoire est un processus fiable qui enregistre tous les messages échangés entre deux processus MPI. Il n'y a donc pas de communication directe entre les processus MPI. MPICH-V2 [42] implémente un protocole de journalisation pessimiste à l'émission. Il améliore les performances de MPICH-V1 en enregistrant les messages sur l'émetteur grâce à un processus *Event Logger*. MPICH-Cl [44] offre un protocole de sauvegarde coordonnée non bloquante qui utilise un serveur de sauvegarde fiable pour sauvegarder. MPICH-VCl [102] est une amélioration de MPICH-Cl. Chaque processus conserve une copie locale de la sauvegarde émise sur le serveur de sauvegarde. Ceci permet d'accélérer le redémarrage en cas de panne. MPICH-VCausal [102] implémente un protocole de journalisation causale optimisée. Il propose de supprimer le principal inconvénient des protocoles de journalisation causale (le volume important d'informations de causalité aggrégées à chaque message) en les sauvegardant de manière asynchrone sur un processus fiable *Event Logger*. MPICH-Pcl [56] fait partie de la nouvelle version de MPICH-V. Elle est basée sur MPICH-2 qui supporte le standard MPI-2. C'est un protocole de sauvegarde coordonnée bloquante.

Tous ces protocoles ont été comparés les uns aux autres dans [43, 102, 44, 56]. De ces études, il en ressort principalement les conclusions suivantes. Les protocoles par sauvegarde coordonnée offrent de bonnes performances comparés aux protocoles par journalisation pour des exécutions sans panne, mais également pour des exécutions en présence de pannes. La coordination n'est pas le premier facteur limitant des protocoles par sauvegarde coordonnée ; la dégradation des performances est principalement due à la charge des serveurs de sauvegarde lors des phases de sauvegarde et de reprise. Enfin, lorsque la fréquence d'apparition des défaillances devient importante, les protocoles par journalisation sont plus intéressants puisqu'ils permettent d'avancer dans le calcul.

Intergiciel	Sauvegarde locale	Protocole	Composants de stockage	Composants fiables	Processeur de rechange	Type de reprise
CoCheck (1996) [136]	Espace mémoire (Condor)	Sauvegarde coordonnée bloquante	Serveurs de sauvegarde	Serveurs de sauvegarde	Nécessaire	Globale
MPICH-Cl (2003) [44]	Espace mémoire (Condor)	Sauvegarde coordonnée non bloquante	Serveurs de sauvegarde	Serveurs de sauvegarde + 1 Dispatcher + 1 Checkpoint Scheduler	Nécessaire	Globale
MPICH-Vcl (2003) [102]	Espace mémoire (Condor, Libckpt ou BLCR)	Sauvegarde coordonnée non bloquante	Machine locale + Serveurs de sauvegarde	Serveurs de sauvegarde + 1 Dispatcher + 1 Checkpoint Scheduler	Nécessaire	Globale
FTC-Charm++ (2004) [164]	Niveau intergiciel	Sauvegarde coordonnée bloquante	Machine locale + Buddy processor	-	Non nécessaire	Globale
MPICH-Pcl (2006) [56]	Espace mémoire (Condor, Libckpt ou BLCR)	Sauvegarde coordonnée bloquante	Machine locale + Serveurs de sauvegarde	Serveurs de sauvegarde + 1 processus mpiexec	Nécessaire	Globale
Open MPI (2007) [90]	Espace mémoire (BLCR) ou fonctions de l'utilisateur	Sauvegarde coordonnée bloquante	Serveurs de sauvegarde	Serveurs de sauvegarde	Nécessaire	Globale
Kaapi-CCK (2008)	Niveau intergiciel	Sauvegarde coordonnée bloquante	Serveurs de sauvegarde	Serveurs de sauvegarde	Non nécessaire	Globale ou partielle
Kaapi-TIC (2005) [93]	Niveau intergiciel	Sauvegarde induite par les communications	Serveurs de sauvegarde	Serveurs de sauvegarde	Non nécessaire	Locale
MPICH-V1 (2002) [40]	Espace mémoire (Condor)	Journalisation pessimiste	Serveurs de sauvegarde + Canaux mémoire	Serveurs de sauvegarde + Cannaux mémoire + 1 Dispatcher	Nécessaire	Locale
MPICH-V2 (2003) [42]	Espace mémoire (Condor)	Journalisation pessimiste à l'émission	Serveurs de sauvegarde + Event Logger	Serveurs de sauvegarde + Event loggers + 1 Dispatcher + 1 Checkpoint Scheduler	Nécessaire	Locale
MPICH-VCausal (2004) [102]	Espace mémoire (Condor)	Journalisation causale à l'émission	Serveurs de sauvegarde + Event Logger	Serveurs de sauvegarde + Event loggers + 1 Dispatcher + 1 Checkpoint Scheduler	Nécessaire	Locale
FTL-Charm++ (2004) [47]	Niveau intergiciel	Journalisation pessimiste à l'émission	Machine locale + Buddy processor	-	Nécessaire	Locale
Satin (2006) [159]	-	Protocole *Ad hoc* utilisant une table globale	-	-	Non nécessaire	-

Tab. 2.2: Comparaison des principales implémentations de protocoles de tolérance aux fautes

2.5.3.2 Charm++

Charm++ est un langage parallèle orienté objet qui offre une virtualisation des processeurs. L'application est écrite en utilisant des objets C++ spéciaux, appelés *chares*. Les méthodes des *chares* peuvent être appelées de manière asynchrone à partir des autres *chares*.

Les *chares* peuvent être migrés entre processus ce qui permet à Charm++ d'offrir un équilibrage de charge automatique à l'exécution. Grâce à cette propriété, Charm++ n'a pas besoin de processus de remplacement. En effet en cas de panne, les processus défaillants peuvent être restaurés sur des processus non défaillants, puis la charge entre les processus est équilibrée automatiquement.

Charm++ propose deux protocoles de tolérance aux fautes. FTC-Charm++ [164] implémente une sauvegarde coordonnée bloquante. Le choix est de ne reposer sur aucun composant fiable, donc deux copies de chaque sauvegarde sont conservées, une localement et l'autre sur un autre processus de calcul, appelé *Buddy processor*. Avec cette méthode, FTC-Charm++ ne peut pas tolérer tous les types de défaillances, en particulier lorsqu'un grand nombre de machines tombe en panne. Néanmoins, les auteurs argumentent du bien fondé de l'approche en estimant en pratique rare les cas où le système ne peut redémarrer.

FTL-Charm++ [47] utilise un protocole de journalisation pessimiste à l'émission. Si un message est émis vers un *chare* distant (*i.e.* sur un processus distant), ce message est sauvegardé sur l'émetteur. Si un message est émis vers un *chare* local (*i.e.* sur le même processus), ce message est sauvegardé sur un autre processus (le *buddy processor*). Un numéro est associé à chaque message, ce qui permet de rejouer les messages dans le même ordre à la reprise.

Adaptive MPI (AMPI) est une implémentation de MPI qui repose sur Charm++. Elle bénéficie donc automatiquement des propriétés d'équilibrage de charge et de tolérance aux fautes de Charm++.

2.5.3.3 Open MPI

Open MPI est une implémentation qui supporte entièrement le standard MPI-2. L'architecture de la partie tolérance de fautes a été conçue pour être flexible et modulaire de manière à encourager l'expérimentation de nouvelles techniques. Elle est découpée en cinq composants [90].

- *Snapshot Coordinator* : il est responsable de lancer, surveiller et collecter les demandes de sauvegarde.
- *File Management* : il gère les fichiers liés aux sauvegardes.
- *Distributed Checkpoint/Restart Coordination Protocol* : ce composant s'occupe du protocole de coordination qui garantit que l'état global est cohérent. Actuellement, seule une sauvegarde coordonnée bloquante similaire à celle de LAM/MPI est implémentée.
- *Local Checkpoint/Restart System* : il est responsable de sauvegarder et de restaurer l'état local des processus. Pour le moment, il supporte la sauvegarde au niveau de l'application grâce à une API qui permet à l'utilisateur de spécifier ses fonctions de sauvegarde et de restauration et la sauvegarde sous forme d'espace mémoire grâce à BLCR.

- *MPI Library Notification Mechanisms* : ce composant informe et coordonne les autres parties de l'implémentation MPI des évènements de sauvegarde et de reprise.

2.5.3.4 Satin

Satin est un environnement de programmation parallèle en Java basé sur le principe « diviser pour régner ». Avec Satin, les processeurs ont des files de travail qui contiennent des tâches qui représentent le travail à exécuter. La charge est équilibrée en utilisant un ordonnancement par vol de travail. Une caractéristique du modèle de programmation de Satin est que les tâches n'ont pas d'effet de bord.

Le protocole de tolérance aux fautes proposé dans Satin ne repose pas sur les méthodes classiques comme la sauvegarde ou la journalisation. La tolérance aux fautes est spécialisée pour l'ordonnancement par vol de travail [159]. Le protocole fonctionne de cette manière : en cas de panne, le résultat d'une tâche orpheline (*i.e.* volée à un processeur défaillant) est enregistré dans une table globale si la tâche est finie ; sinon la tâche orpheline est annulée. Concernant les tâches volées par un processeur défaillant, elles sont réordonnancées. Les résultats de la table globale peuvent-être alors utilisés au lieu d'exécuter une tâche.

Cette méthode garantit que le calcul se terminera et elle réduit la quantité de travail perdu parce que l'exécution utilisera les résultats de la table globale après une défaillance. Cette méthode ne nécessite pas de composant fiable ni de processeur de remplacement. Mais en cas de panne de tous les processus, l'application devra être réexécutée entièrement.

Néanmoins, la mise en route du protocole n'intervient qu'après la première panne. La quantité de travail perdu peut donc être assez importante [92].

2.5.3.5 Kaapi

Kaapi est un moteur d'exécution qui permet d'exécuter des applications parallèles qui sont écrites à l'aide du langage Athapascan. Athapascan permet de décrire une application sous-forme d'un graphe de flot de données de manière indépendante de la plate-forme d'exécution. Ce graphe de flot de données est utilisé pour la représentation interne de l'état de l'application dans Kaapi. Le moteur d'exécution Kaapi propose deux méthodes d'ordonnancement pour exécuter le graphe de flot de données, le vol de travail et le partitionnement statique.

Dans Kaapi, la sauvegarde de l'état des processus est réalisé au niveau intergiciel grâce à la représentation abstraite de l'application sous forme d'un graphe de flot de données. Grâce à cette représentation, l'état de l'application est portable (il est possible de redémarrer sur une architecture différente) et manipulable (il est possible de fusionner, découper le calcul entre plusieurs). Ainsi en cas de défaillance, l'application peut redémarrer sans machine de remplacement. Les algorithmes d'ordonnancement permettent alors d'équilibrer la charge pour continuer l'exécution de manière efficace.

Kaapi est l'environnement qui a été utilisé pour réaliser les expérimentations présentées dans ce manuscrit. Il propose deux protocoles de tolérance aux fautes, TIC et CCK.

- Le protocole TIC (*Theft-Induced Checkpointing*) est basé sur le protocole CIC (*Communication-Induced Checkpointing*) présenté à la section 2.4.2.3). Cependant, il a été optimisé et spécialisé pour le modèle graphe de flot de données et une exécution avec vol de travail. Après défaillance, ce protocole permet de reprendre l'exécution de l'application en effectuant seulement la reprise des processus défaillants.
- Le protocole CCK (*Coordinated Checkpointing in* Kaapi) est basé sur la technique de sauvegarde coordonnée à la section 2.4.2.2. Cependant il propose, en plus de la reprise globale, une reprise partielle où seul le travail strictement nécessaire est réexécuté pour redémarrer l'application. Ce protocole fait partie des travaux présentés dans cette thèse, il est détaillé au chapitre 9.

2.6 Conclusion

Ce chapitre a présenté un aperçu de la tolérance aux fautes pour les systèmes distribués de calcul haute performance.

En premier lieu, nous avons abordé d'un point de vue très général la sureté de fonctionnement et le vocabulaire associé au domaine de la tolérance aux fautes. Il apparait que la tolérance aux fautes n'est qu'une des approches qui permettent d'assurer la sureté de fonctionnement.

Les techniques de tolérance aux fautes spécifiques aux systèmes distribués sont basées sur la réplication ou sur l'utilisation d'une mémoire stable. La réplication n'est pas adaptée au domaine du calcul haute performance puisqu'elle nécessite un nombre important de ressources qui pourraient être utilisées pour accélérer le calcul.

Dans le cadre de cette thèse, nous nous sommes donc limités aux techniques de tolérance aux fautes basées sur l'utilisation d'une mémoire stable. La difficulté de ces techniques repose sur la construction (à la sauvegarde ou la reprise) d'un état global cohérent de l'application.

Les protocoles de reprise basés sur la sauvegarde visent à construire un état global cohérent, c'est-à-dire un état qui aurait pu se produire durant une exécution sans panne. Pour cela, ils cherchent à déterminer un ensemble de sauvegardes qui forme un état exempt de message orphelin.

Les protocoles de reprise basés sur la journalisation font l'hypothèse d'une exécution déterministe par morceaux (hypothèse PWD) et visent à rétablir l'état de l'application exactement tel qu'il était avant la panne (cet état est donc nécessairement cohérent). Pour cela, ces protocoles rejouent les évènements non déterministes sauvegardés qui ont amené les processus à cet état.

Chaque protocole possède ses propres caractéristiques et le choix d'un protocole de tolérance aux fautes doit être guidé par les propriétés de l'application, par la plate-forme d'exécution et par la fréquence d'apparition des pannes.

La recherche dans le domaine de la tolérance aux fautes pour les systèmes distribués est déjà ancienne ; le nombre d'implémentations réalisées est assez conséquent. Les travaux les plus récents et les plus représentatifs ont été présentés. Il est à noter que les protocoles les plus utilisés sont les protocoles les plus simples.

Adaptation et reconfiguration dynamique 3

Sommaire

3.1 Introduction

Comme cela a été évoqué dans l'introduction, les architectures de calcul (multi-cœurs, grilles de calcul) sont des plates-formes d'exécution très dynamiques. Cet aspect dynamique prend plusieurs formes : apparation ou disparition de ressources, charge et vitesse variables des processeurs et des réseaux, matériels et logiciels hétérogènes. Dans un tel contexte, il apparait nécessaire d'adapter l'application pour l'exécuter avec les meilleures performances.

Les notions d'adaptation et de reconfiguration seront détaillées dans la suite de ce chapitre, mais nous pouvons déjà préciser qu'elles sont liées. L'adaptation est le

processus qui va permettre, en observant l'application et l'environnement d'exécution, de choisir la configuration qui correspond le mieux[1]. La reconfiguration est alors l'étape qui consiste à appliquer la nouvelle configuration choisie.

L'adaptation et la configuration **statiques** sont celles qui sont réalisées avant ou au début de l'exécution de l'application. Par exemple, l'adaptation statique peut se faire à la compilation ; la reconfiguration prend alors la forme d'une recompilation. Mais elle peut aussi se faire au lancement de l'application, par l'intermédiaire de paramètres ou de fichiers de configuration. Dans tous les cas, cela nécessite de connaitre le contexte d'exécution au moment de l'adaptation statique. Une telle adaptation n'aura donc du sens que si ce contexte ne varie pas (ou peu) lors de l'exécution.

Si l'adaptation et la reconfiguration sont effectuées au cours de l'exécution de l'application, on dira qu'elles sont **dynamiques**. Une telle adaptation peut intervenir plusieurs fois au cours de l'exécution d'une application. Ce cas est adapté lorsque le contexte d'exécution peut changer à tout moment. L'application requiert dans ce cas un mécanisme de reconfiguration dynamique, ce qui va accroitre sa complexité. Par la suite, nous nous intéressons principalement à l'adaptation et à la reconfiguration dynamique.

Ce chapitre est organisé en trois sections. La section 3.2 présente la notion d'adaptation dynamique du point de vue général des systèmes informatiques. La section 3.3 détaille le problème de la reconfiguration dynamique pour les applications distribuées. Enfin, la dernière section détaille le cas plus spécifique des applications parallèles pour le calcul haute performance.

3.2 Adaptation dynamique

3.2.1 Problématique

Pour qu'un système informatique fournisse un service adapté[1], il est nécessaire de le configurer correctement. Dans les cas simples, la configuration peut être réalisée manuellement. Dans le cas d'applications complexes, l'utilisateur n'a pas forcément les connaissances nécessaires pour choisir les meilleurs paramètres ou la meilleure configuration. De plus, si l'environnement d'exécution de l'application a des caractéristiques variables, il peut être nécessaire de réajuster ces paramètres régulièrement.

Le but de l'adaptation dynamique est de pallier ce problème en proposant un mécanisme pour définir (et redéfinir) la configuration de l'application. Le choix de la bonne configuration dépend évidemment du critère d'évaluation. Au final, pour une application et un contexte d'exécution donnés, il est nécessaire de se poser les questions suivantes.

Quels sont les critères à optimiser ? Ceci permet de définir le but de l'adaptation. Il dépend évidemment du type d'application et des besoins de l'utilisateur. Par exemple, dans le cas du calcul haute performance, le critère qui est généralement considéré est le temps d'exécution. Le but de l'adaptation va donc être de trouver la configuration qui

[1] Cela nécessite évidemment un critère d'évaluation.

donnera le temps d'exécution le plus court. Dans le cas d'une application interactive, le critère pourra être le temps de réactivité de l'application.

À quels changements souhaite-t-on s'adapter ? L'adaptation est nécessaire parce que les performances de l'application (au sens défini ci-dessus) ne sont pas toujours optimales selon le contexte d'exécution. Les changements à prendre en compte sont généralement ceux qui affectent les performances de l'application et pour lesquels la modification de la configuration pourrait améliorer les performances. Par exemple, il sera intéressant d'adapter une application parallèle à l'ajout de machines s'il est possible d'extraire plus de parallélisme et de l'exécuter sur ces processeurs supplémentaires. Une classification des changements possibles du contexte d'exécution est présentée à la section 3.2.3.

Quels sont les changements de configuration possibles ? Les changements de configuration possibles dépendent de l'application. Les différentes configurations peuvent s'exprimer de plusieurs manières : changement de la valeur d'un paramètre, changement de la méthode de calcul, changement du protocole de communication, etc. Certaines configurations auront plus ou moins d'influence sur les performances de l'application. Dans le cas d'une application parallèle, un changement de configuration pourra correspondre à exposer plus de parallélisme pour permettre une exécution sur plus de machines.

Comment choisir la nouvelle configuration en fonction des nouvelles contraintes ? Le choix de la nouvelle configuration doit être fait pour améliorer les performances de l'application. La décision est prise en tenant compte de plusieurs éléments : les critères à optimiser, les nouvelles contraintes d'exécution et les changements de configuration possibles. Les différents mécanismes existants seront présentés en section 3.2.4.3.

3.2.2 Définitions et terminologie

Dans cette section, nous fixons la terminologie et définissons les principales notions qui seront utilisées dans le reste de ce document. Ces définitions sont partiellement extraites de [57].

Le **système informatique** considéré correspond à l'ensemble des éléments que l'on cherche à optimiser et qui sont la cible de l'adaptation. Les autres éléments qui interagissent avec le système constituent le contexte d'exécution.

Le **contexte d'exécution** correspond à l'ensemble des éléments qui influencent le système lors de son exécution. Ce contexte prend en compte à la fois l'environnement physique (matériel et logiciel) mais aussi l'attente des utilisateurs. Le contexte d'exécution peut être très dynamique. Les différentes causes de **variabilité** du contexte d'exécution sont présentées à la section 3.2.3.

Le **critère de performance** est ce qui permet d'évaluer l'adéquation du système vis-à-vis de son contexte d'exécution. Il est lié à la fonction du système et au besoin de l'utilisateur. Il peut être composé de plusieurs objectifs.

Une **adaptation** est une modification du système, en réponse à un changement dans son contexte, dont l'objectif est de rendre le système résultant plus performant

(au sens du critère de performance choisi) dans le nouveau contexte. L'adaptation est composée de trois fonctions :

- L'**observation** consiste à récupérer des informations sur le contexte d'exécution. Elle est détaillée à la section 3.2.4.1.
- La **reconfiguration** correspond à l'étape modification du système pour le passer de son ancienne configuration à sa nouvelle configuration. Elle est présentée à la section 3.2.4.2 puis détaillée en 3.3.
- La **décision** consiste à choisir la meilleure reconfiguration, en fonction du critère de performance considéré et en fonction du contexte d'exécution observé. Elle est détaillée en section 3.2.4.3

3.2.3 Variabilité du contexte d'exécution

Les variations des caractéristiques de l'environnement ont plusieurs origines. Dans [57], David les classe en trois types : les variabilités spatiales qui sont liées à la diversité des plates-formes d'exécution ; les variabilités temporelles qui sont dues à la dynamicité des systèmes ; et les variabilités des besoins qui correspondent à la diversité des utilisateurs et des utilisations. La suite de cette section présente les différents types de variabilités qui existent en les illustrant par quelques exemples sans l'intention d'être exhaustif.

Les plates-formes d'exécution sont très variées. Les caractéristiques techniques peuvent être très différentes entre le circuit logique programmable (FPGA[2]), les systèmes multiprocesseurs sur puce (MPSoC[3]), les téléphones portables, les ordinateurs de bureau, les consoles de jeu et les supercalculateurs. Bien que ces plates-formes répondent au départ à un besoin spécifique, les frontières entre les cas d'utilisation sont de plus en plus floues et il n'est pas rare qu'une application soit destinée à plusieurs types de plates-formes. Les caractéristiques techniques varient notamment du point de vue de l'architecture du processeur, de la vitesse, du système d'exploitation, des périphériques physiques, etc. L'hétérogénéité entre ces plates-formes apparait également au niveau de leurs capacités (puissance de calcul, mémoire disponible, connectivité).

Enfin, l'application peut également être distribuée sur plusieurs machines. Ces architectures distribuées peuvent prendre plusieurs formes : grappe et grilles de calcul, réseaux pair-à-pair, réseaux domestiques, etc. La complexité est encore augmentée lorsque les machines qui composent cette plate-forme distribuée sont hétérogènes. Un exemple marquant est la plate-forme BOINC [11] qui regroupe à la fois les ressources des ordinateurs de bureau, des consoles de jeu PlayStation 3 et des cartes graphiques.

Un autre aspect est celui de la variabilité temporelle. En effet, les caractéristiques de la plate-forme d'exécution peuvent varier au cours du temps. Par exemple sur un ordinateur portable, la fréquence du processeur peut changer en fonction du niveau de sa batterie. Les téléphones portables (et tous les appareils mobiles en général) ont des contraintes sur les réseaux de communication (Wifi, 3G, GPRS, etc. ou même aucun réseau disponible) qui influencent notamment le débit et la latence des communications. Les nœuds d'une grille de calcul peuvent voir leur charge évoluer en fonction des activités

[2] *Field-Programmable Gate Array*
[3] *MultiProcessor System-on-Chip*

des utilisateurs. Les plates-formes d'exécution distribuées ajoutent une dynamicité supplémentaire puisque le nombre de machines disponibles peut également changer. La variabilité temporelle peut aussi avoir pour origine l'application elle-même. Par exemple, une application de simulation peut avoir une première étape de calcul intensif, puis une seconde étape de visualisation des résultats moins gourmande en calcul.

Enfin, le besoin de l'utilisateur peut changer pour une même application. Une application de simulation qui fonctionne en mode interactif aura tendance à réduire la qualité de la simulation pour offrir un temps de réaction plus rapide. Les téléphones portables proposent un mode silencieux qui coupe le son de toutes les applications. La qualité d'un encodage vidéo peut être adaptée au type d'appareil qui sera utilisé pour le visionner (téléphone portable, ordinateur de bureau ou projecteur de cinéma).

3.2.4 Mécanique de l'adaptation

Les trois fonctions essentielles à la mécanique de l'adaptation sont l'**observation**, la **décision** et la **reconfiguration**. Les travaux qui cherchent à modéliser le processus d'adaptation de manière générale sans s'attacher à une application précise [45, 69, 4, 52] font une séparation claire de ces trois fonctions. Dans les travaux spécifiques à une application ou à un modèle d'exécution [143, 158, 2], ces fonctions apparaissent de manière plus implicite.

La suite de cette section précise le rôle de chacune de ces fonctions et donne les différentes approches utilisées pour les réaliser.

3.2.4.1 Observation

La fonction d'observation va permettre au système de connaitre l'état de l'environnement d'exécution. L'adaptation se faisant en fonction du contexte d'exécution, il est indispensable de connaitre son état et ses variations [62].

De manière générale, l'observation peut fournir plusieurs services :

- La *notification* permet à un composant de demander la surveillance de certains paramètres et d'être averti en cas de changement.
- L'*interrogation* permet à toute autre partie du système, notamment les fonctions de décision et de reconfiguration, de prendre connaissance de l'état du système pour réaliser leur action correctement.
- La *prédiction* essaie d'anticiper les évolutions à venir à l'aide des mesures effectuées.

Les logiciels dédiés à l'observation offrent une interface pour accéder à leurs services. WildCAT [58, 57] et l'architecture GMA [139], implémentée dans Mercury [81], supportent à la fois l'interrogation et la notification. Network Weather Service [156] et Delphoi [106] ne proposent pas de service de notification mais offrent l'interrogation et la prédiction.

En dehors de ces logiciels dédiés à l'observation, de nombreux travaux sur l'adaptation utilisent leur propre système de surveillance. On peut distinguer ceux qui offrent un cadre généraliste, en proposant de surveiller tous types de paramètres comme dans ACEEL [51], Dynaco [45] ou PCL [69]. À l'opposé, les travaux spécifiques à une application ou à un modèle d'exécution surveillent des caractéristiques précises du contexte d'exécution, comme l'ajout ou la disparition de machine dans Starfish

MPI [2], ou la charge des processeurs et les temps d'inactivité des processus dans AMPI [87].

Enfin, on trouve aussi une autre approche dans GrADS [143] et ASSIST [5] qui utilise la notion de contrat. Le contrat spécifie les attentes en termes de performances. Le composant d'observation surveille alors l'application et lorsque le contrat est rompu, l'adaptation est déclenchée.

3.2.4.2 Reconfiguration

La fonction de reconfiguration est celle qui va modifier la configuration du système de manière à ce qu'il soit mieux adapté au contexte d'exécution. La reconfiguration dynamique est une étape particulièrement critique puisqu'elle se déroule pendant l'exécution de l'application. Le processus d'adaptation (et donc *a fortiori* la reconfiguration) ne doit pas entraver le fonctionnement de l'application. Il est nécessaire de trouver à quel moment l'exécuter et aussi comment appliquer correctement les modifications. Ce problème est exacerbé dans le cas des applications distribuées. La description du processus de reconfiguration dans cette section est succincte. La section 3.3 aborde le problème de la reconfiguration pour les applications distribuées de manière plus détaillée.

On peut déjà distinguer les travaux en deux catégories, ceux qui fixent le type de reconfiguration et ceux qui proposent un canevas générique permettant d'appliquer tout type de reconfiguration. Dans GrADS [143], Starfish MPI [2], AMPI [87], Satin [158] et ASSIST [147] les reconfigurations proposées se limitent à la migration de processus et à l'ajout et la suppression de processus. D'autres travaux s'attachent à reconfigurer la couche de communication [149, 148].

Des modèles de reconfiguration plus généraux reposent généralement sur un des paradigmes suivants [132] : un modèle par composants, la reflexion, la programmation par aspect ou bien les patrons de conception. Ce sont ces outils qui permettent de modifier simplement la configuration de l'application et d'apporter certaines garanties sur la cohérence. Dans ACEEL [51], les composants adaptables doivent suivre le patron de conception Strategy. Safran [57, 59] propose des actions permettant de modifier la structure des composants et offre aussi un protocole à méta-objets. Dans [45], Buisson propose la notion de plan. La reconfiguration est réalisée en exécutant le plan qui contient une liste d'instructions spécifiques sur lesquelles il est possible de raisonner. PCL [69] offre la reflexion du programme de l'application en utilisant une représentation sous forme d'un graphe statique de tâches (*Static Task Graph*). Les reconfigurations sont alors décrites à l'aide d'opération sur ce graphe.

3.2.4.3 Décision

La fonction de décision est un point clé dans le processus d'adaptation. L'objectif de la décision est de choisir une reconfiguration adéquate en fonction du contexte d'exécution. Elle est « l'intelligence » de l'adaptation. La fonction de décision repose donc sur la fonction d'observation pour connaitre l'état du contexte d'exécution. L'adéquation de la configuration choisie est liée au critère de performance considéré.

De nombreux travaux visent un type de reconfiguration fixé. Dans ces cas là, la fonction de décision est donnée avec l'application par le biais d'un modèle de

performance de l'application. Ce modèle définit implicitement le critère de performance et les informations observées. C'est ce qui est fait dans GrADS [143] et ASSIST [147]. L'approche de AMPI est très similaire puisqu'elle emploie des stratégies, qui ne sont pas spécifiques à une application, mais plutôt à un type d'application [87]. Au contraire, Satin propose un algorithme de décision basé sur un modèle d'exécution indépendant d'une application donnée [158].

Dans le cas d'un moteur d'adaptation générique, ni le critère de performance, ni les informations observées, ni les reconfigurations possibles ne sont définis. La réalisation de la fonction de décision est alors un problème complexe. Il faut donner un moyen d'exprimer le choix de la reconfiguration.

Dans PCL, la décision est gérée dans une fonction appelée *Adaptor*. Elle est écrite dans un langage de programmation classique [68]. Safran [57] et ACEEL [51] proposent de définir les politiques d'adaptation sous forme de règles basées sur le paradigme ECA (évènement, condition, action). Lorsque l'évènement se produit, la condition est testée et, si elle est vérifiée, l'action de reconfiguration sera exécutée. VGrADS (qui étend le projet GrADS [23]) utilise la logique floue pour évaluer le contrat de performance donné par l'application [127]. Enfin, [114] présente une approche probabiliste au problème de la décision en couplant un modèle de décision Bayésien et un modèle de décision Markovien.

Dans [45], Buisson ajoute une fonction supplémentaire : la planification. Dans ses travaux, le choix de la nouvelle reconfiguration se fait en deux étapes : la décision telle que nous l'avons présentée et la planification qui consiste à déterminer par quels moyens adopter la nouvelle configuration. Pour cela, la planification utilise un programme *guide* qui est spécifique à l'entité à adapter et construit le plan de reconfiguration. C'est ce plan qui est exécuté lors de l'étape de reconfiguration.

3.3 Reconfiguration dynamique des applications distribuées

Dans la section précédente, nous avons présenté dans quel contexte pouvait s'utiliser la reconfiguration dynamique. Cette nouvelle section nous permet d'aborder les problèmes essentiels qui touchent la reconfiguration dynamique d'applications distribuées. Nous présentons d'abord des critères de classification puis les propriétés des reconfigurations. Ensuite, nous détaillerons les problèmes de cohérence liés à une reconfiguration.

3.3.1 Classification

Plusieurs critères ont été proposés pour classer les reconfigurations [6, 110, 84]. Nous allons nous intéresser aux critères suivants : type de reconfiguration et origine de la reconfiguration.

Dans [85, 84], Hofmeister modélise une application distribuée comme un ensemble d'entités qui contiennent des programmes et des données. Chaque entité expose aux autres entités une interface. Les connexions entre les interfaces représentent les canaux

de communication entre les entités. L'ensemble des entités et des connexions forment la **structure** de l'application. La **géométrie** de l'application décrit comment cette structure est déployée sur la plate-forme d'exécution.

Hofmeister distingue trois types de reconfiguration.

- Les **changements geométriques** sont les reconfigurations qui ne modifient pas la structure de l'application mais qui changent le placement des entités sur la plate-forme d'exécution. Ce type de changements ne modifie pas le degré de parallélisme de l'application. C'est typiquement le cas de la migration d'un processus. Ces changements utilisent généralement les techniques de sauvegarde/reprise du domaine de la tolérance aux fautes : l'exécution est arrêtée ; l'état du processus est capturé et transféré sur la nouvelle machine ; puis un processus est démarré sur la nouvelle machine et reprend le calcul.
- Les **changements structurels** sont les reconfigurations qui vont modifier les connexions entre les entités, ou bien ajouter ou supprimer des entités. Par exemple, ce type de modifications permet de changer le parallélisme de l'application. Il est plus complexe puisque les opérations nécessaires à la reconfiguration dépendent de l'application. Généralement, une telle reconfiguration ne peut s'appliquer qu'à certains points précis de l'exécution de manière à garantir une certaine cohérence.
- Les **remplacements d'entités** consistent à changer l'implémentation interne d'une entité sans changer la structure de l'application. Ceci représente le cas où l'algorithme de calcul est changé pour réduire le temps de calcul (au détriment de la précision du calcul par exemple) sans modification de l'interface de l'entité. Ce cas a été étudié en détails dans [36, 37]

Dans [110], Goudarzi propose de distinguer les changements programmés et les changements évolutifs.

- Les **changements programmés** (*programmed changes*) sont ceux qui ont été prévus par le développeur et qui ont donc été pris en compte lors de la conception de l'application.
- Les **changements évolutifs** (*evolutionary changes*) sont les changements imprévisibles ou imprévus au moment de la conception mais qui deviennent nécessaires au cours de la durée de vie de l'application.

Bien que les changements programmés puissent être réalisés par des mécanismes *ad hoc* de l'application, les changements évolutifs nécessitent des mécanismes plus généraux, notamment en faisant apparaitre une certaine abstraction de l'état de l'application [147].

3.3.2 Propriétés de la reconfiguration

[147] identifie plusieurs propriétés du processus de reconfiguration. Les propriétés indispensables sont la cohérence et la généralité.

- La **cohérence** signifie qu'une reconfiguration doit laisser l'application dans un état correct. La problématique de la cohérence est détaillée en 3.3.3.
- La **généralité** indique que le processus de reconfiguration doit pouvoir supporter tous les types de reconfigurations sur tous les types d'entités.

Les propriétés souhaitables sont la scalabilité et l'efficacité.

- La **scalabilité** indique que la reconfiguration doit pouvoir s'appliquer à tout le système ou seulement à une petite partie. Dans le cas où seulement une partie

du système est concernée par la reconfiguration, le reste du système doit pouvoir continuer de fonctionner pendant la reconfiguration.
- L'**efficacité** signifie que le temps de reconfiguration doit être aussi petit que possible de manière à réduire l'interruption de service.

Enfin les deux propriétés importantes pour faciliter l'utilisation du mécanisme d'adaptation sont la responsabilité de la cohérence et la transparence.

- La **responsabilité de la cohérence** d'une reconfiguration ne devrait pas être laissée à l'utilisateur puisque cela nécessite une certaine connaissance du fonctionnement interne de l'application ou du moteur d'exécution. Laisser la responsabilité de la cohérence au mécanisme d'adaptation permet de concevoir des reconfigurations indépendantes de l'application et du moteur d'exécution.
- La **transparence** permet à l'utilisateur de ne pas se soucier de la sémantique de la reconfiguration lors de la programmation de l'application.

3.3.3 Cohérence de la reconfiguration

La reconfiguration est un processus qui peut modifier l'application en profondeur. Elle peut mettre en œuvre la création, la suppression ou la migration d'entités et interférer avec les interactions en cours entre les entités. Le processus de reconfiguration ne doit pas entrainer d'erreurs entre les entités qui interagissent.

La cohérence est une propriété indispensable d'une reconfiguration car le système peut devenir inutilisable si elle n'est pas respectée. Le système doit se retrouver dans un état correct après une reconfiguration pour prévenir une défaillance. Pour garantir la cohérence d'un système après une reconfiguration, les trois aspects suivants ont été identifiés [110, 6].

- Le système doit satisfaire ses obligations d'**intégrité structurelle** (*structural integrity*).
- Les entités du système doivent avoir des **états mutuellement cohérents** (*mutually consistent states*).
- Les **invariants d'état de l'application** doivent être vérifiés (*application state invariants*).

D'autres travaux identifient des propriétés similaires [61, 72].

Intégrité structurelle. La structure du système doit respecter les contraintes des interfaces des entités et la manière dont elles sont connectées.

Au niveau objet (au sens programmation orientée objet), cela peut être garanti grâce aux techniques d'héritage ou vérifié par typage dynamique. Au niveau composant, le modèle de composants doit offrir des moyens de vérifier les contraintes de l'interface à l'exécution. Cela nécessite un moyen d'exprimer ces contraintes [162, 153].

Cohérence mutuelle. Les interactions sont le seul moyen par lequel une entité peut en affecter une autre. Des entités sont dites être dans des états mutuellement cohérents si chaque interaction résulte en un état cohérent une fois terminée [7, 98, 111, 110]. En conséquence, si deux entités interagissent, elles auront la même perception du résultat de l'interaction (réussie ou échouée).

Pour garantir la cohérence mutuelle, la plupart des approches préconisent d'effectuer les reconfigurations seulement au moment des états appelés **états sains** pour la reconfiguration (*reconfiguration-safe state*). Un tel état signifie que chaque entité participant à la reconfiguration a un état stable, indépendant et accessible, et qu'elle ne participe à aucune interaction.

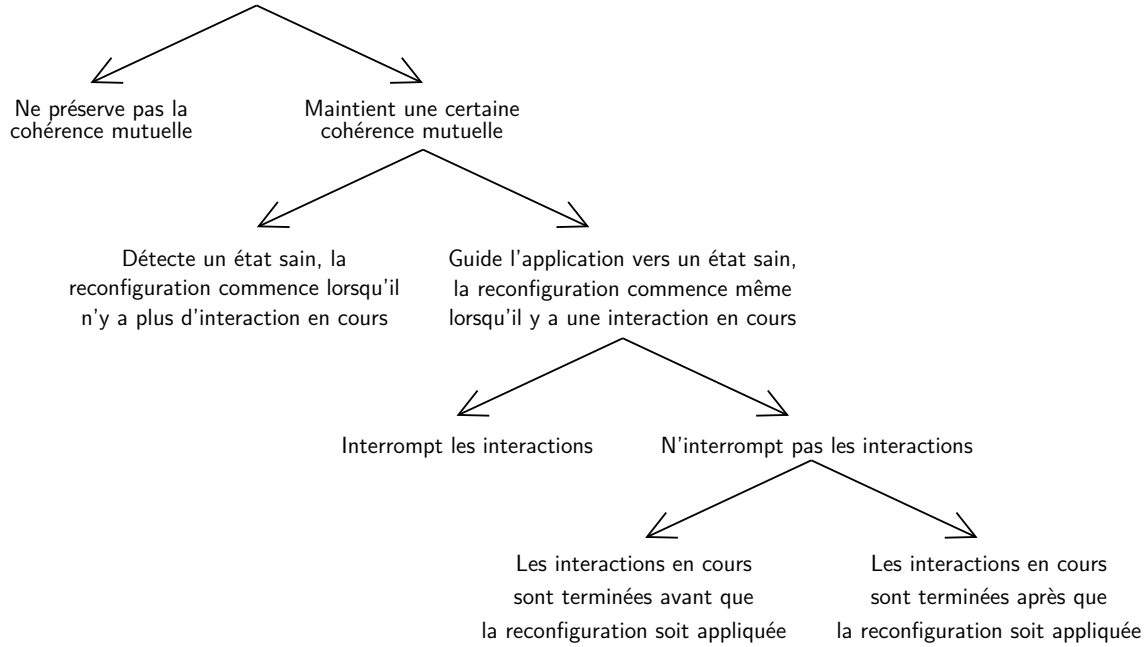


FIG. 3.1: Les approches pour préserver la cohérence mutuelle d'une reconfiguration [6]

Dans [6], Almeida propose une classification des approches visant à garantir la cohérence mutuelle des entités reconfigurées. Cette classification est présentée à la figure 3.1. Parmi les méthodes qui garantissent la cohérence mutuelle des états, Almeida distingue celles qui atteignent un état sain pour la reconfiguration en observant l'exécution du système, et celles qui atteignent un état sain en guidant le système vers cet état. Dans le premier cas, il n'y a pas de garantie qu'un tel état sera atteint et cela dépend du comportement de l'application. Dans le deuxième cas, c'est le rôle du processus de reconfiguration de garantir que cet état sera atteint.

Les approches qui guident l'application pour atteindre un état sain sont séparées en deux catégories [6, 110]. Il y a celles qui interrompent les interactions en cours et les autres. Les processus de reconfiguration basés sur l'interruption des interactions nécessitent un mécanisme de retour en arrière pour reprendre sans erreur en cas d'interruption. Ceux qui n'interrompent pas les interactions sont conçus pour assurer que, une fois la reconfiguration appliquée, les interactions en cours seront terminées.

Parmi cette dernière catégorie, on trouve les travaux [97, 111, 35, 154, 7] qui traitent le problème avec un modèle très général. Ils offrent des solutions assez générales mais également complexes. On peut notamment trouver une comparaison de ces travaux dans [6]. La section 3.4.1 présente les principaux travaux axés sur le domaine du calcul parallèle haute performance.

Invariants d'état de l'application. Les invariants d'état sont des prédicats portant sur tout ou une partie du système. Ces prédicats sont exprimés en fonction des variables d'état du système.

La diversité des reconfigurations possibles ne permet généralement pas au processus de reconfiguration de rétablir les invariants d'état de l'application. Cela nécessite le plus souvent une aide du développeur de l'application. Ce thème est abordé dans [6].

3.4 Cas du calcul parallèle haute performance

Le but de cette section est de détailler les solutions proposées pour fournir des mécanismes d'adaptation et de reconfiguration dynamique pour les applications parallèles de calcul haute performance. Nous nous intéressons particulièrement aux plate-formes d'exécution de type grappes ou grilles de calcul.

Pour ces applications, les critères de performances généralement considérés sont les suivants.

- Le temps d'exécution : on souhaite minimiser le temps pendant lequel l'application s'exécute sur la plate-forme.
- Le temps de complétion : on souhaite minimiser le temps entre la soumission de l'application à la plate-forme et la fin de l'exécution.
- L'efficacité : on souhaite maximiser l'utilisation des ressources de la plate-forme.

Sur les architectures distribuées de type grilles de calcul, le contexte d'exécution a des caractéristiques et des variabilités spécifiques. Les variabilités spatiales sont liées à la plate-forme. Même si nous nous limitons aux grilles de calcul, nous pouvons noter des variabilités spatiales liées à l'hétérogénéité des processeurs (en architecture et en vitesse), des systèmes d'exploitation, des réseaux de communication mais également au niveau de la taille respective des grappes qui composent la grille. Une partie de ces problèmes est réglée par des configurations statiques à la compilation et au lancement de l'application.

Cependant, les variabilités temporelles nécessitent les processus d'adaptation et de reconfiguration dynamique pour être traitées efficacement. Principalement, on trouve l'ajout et la suppression de machines et les variations de charge (processeurs, réseaux, etc.). Les variabilités dues aux besoins des utilisateurs sont très réduites puisque les applications considérées présentent peu d'interactivité.

Concernant la suppression de machines, elle peut avoir plusieurs origines.

Elle peut être volontaire, c'est-à-dire décidée par l'utilisateur ou le gestionnaire de ressources, mais elle peut aussi être liée à une défaillance. La suppression d'une machine de calcul ayant pour cause une défaillance est plus compliquée à gérer puisqu'elle est difficilement prévisible. Nous voyons ici une corrélation importante entre le domaine de l'adaptation et de la tolérance aux fautes. Dans le cas d'une défaillance de machines ; les techniques de tolérance aux fautes peuvent être complétées par un mécanisme d'adaptation pour tenir des machines supprimées. Dans le cas d'une migration de processus, le processus d'adaptation peut requérir des mécanismes de tolérance aux fautes pour pouvoir sauvegarder et reprendre l'exécution d'un processus sur une autre machine.

La suite de cette section présente les principales solutions d'adaptation et de reconfiguration dynamique pour les applications parallèles de calcul haute performance sur grille. Pour cela, nous identifions d'abord les critères considérés pour comparer ces différentes solutions. Ces critères font référence aux principaux thèmes abordés dans ce chapitre.

- **Observation** : indique comment est réalisée la fonction d'observation.
- **Décision** : détaille comment est choisie la nouvelle reconfiguration.
- **Reconfiguration** : donne les types de reconfigurations possibles.
- **Gestion de cohérence** : indique comment cette solution permet de garantir la cohérence de la reconfiguration.

3.4.1 Comparaison des solutions

De très nombreuses solutions fournissent des mécanismes d'adaptation. Il n'est pas possible de toutes les citer. Nous proposons tout d'abord une brève liste des solutions existantes. Puis, la fin de ce chapitre détaille les solutions les plus représentatives et les plus intéressantes pour le domaine du calcul parallèle haute performance.

3.4.1.1 Aperçu

Une partie de ces travaux proposant des mécanismes d'adaptation ou de reconfiguration se limitent à des reconfigurations locales[4]. On peut citer ACEEL [52, 51], Safran [57], DART [82], LEAD++ [10].

D'autres travaux sont dédiés à la gestion des ressources pour les grilles comme Globus [75, 74], AppLes [24] ou GridWay [89].

AFPAC [46] et ASSIST [4] abordent le problème de l'adaptation dynamique de composants parallèles. PCL [69] permet l'adaptation d'applications parallèles en utilisant la réflexion. Satin propose un mécanisme d'adaptation qui se limite à l'ajout et la suppression de processus. De même pour le modèle de programmation MPI, AMPI [87], Starfish MPI [2], GrADS [143] mais aussi [135, 107] considèrent uniquement l'ajout, la suppression ou la migration des processus.

D'autres travaux adaptent le protocole de communication. MPI/CTP [148] repose sur la couche de communication configurable CTP et permet de changer dynamiquement entre les protocoles par *RendezVous* ou *EagerRendezVous* en fonction de la proportion de messages pré-postés. PRO-MPI [149] propose de changer le protocole de communication (*send-receive* ou *RemoteDMA*) avec Infiniband. Pour cela, il détermine, à partir d'exécutions précédentes, des profils d'exécution des différentes phases de l'application.

L'adaptation d'application est également utilisée à des fins de sécurité ou d'administration. Dynasa [134] reprend le modèle d'adaptation de Dynaco [45] pour résoudre les problèmes de sécurité sur les grilles de calcul. Pour cela, il utilise des techniques de tolérance aux fautes (sauvegarde et réplication). Jade [60] est un environnement d'administration autonome qui utilise une mécanique d'adaptation afin de fournir des services d'auto-réparation et d'auto-optimisation.

[4] C'est-à-dire qu'elles ne permettent pas de reconfigurer des objets distribués.

Le domaine du pilotage d'application de simulation numérique aborde aussi le problème de la cohérence. EPSN [70] cherche à appliquer des traitements (observations ou modifications) sur un ensemble temporellement cohérent des données des processus de simulation. Enfin, le domaine de la tolérance aux fautes propose aussi des mécanismes d'adaptation puisqu'au moment de la reprise, il peut être nécessaire de prendre en compte le nouveau contexte d'exécution. Ces environnements sont présentés à la section 2.5.

3.4.1.2 Dynaco / AFPAC

Dans [45], Buisson propose un modèle de canevas d'adaptation dynamique pouvant correspondre à tout type d'adaptation. Cette solution a été implémentée à travers les prototypes Dynaco, AFPAC et Taco. Dynaco est une implémentation du modèle de canevas dynamique pour les composants Fractal et il assure les mécanismes d'adaption. AFPAC assure la gestion de la cohérence de la reconfiguration et Taco permet l'annotation automatique d'une application parallèle pour faire apparaitre son graphe de controle.

Le modèle d'un composant parallèle adaptable avec Dynaco et AFPAC est présenté à la figure 3.2.

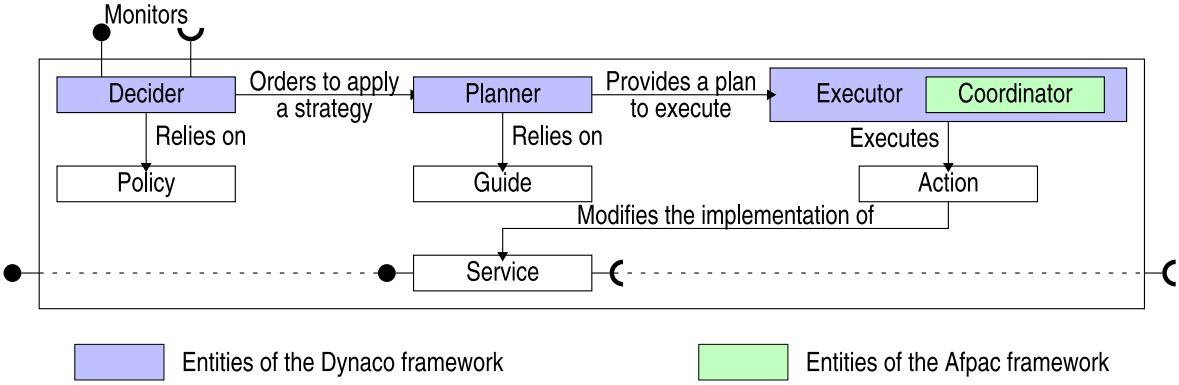


Fig. 3.2: Modèle d'un composant parallèle adaptable avec Dynaco et AFPAC [46]

La solution d'adaptation proposée par Buisson dans [45] repose sur un modèle composant. Un composant adaptable est un composant qui a été étendu pour comporter les fonctions d'observation, de décision, de planification et d'exécution. Un composant adaptable peut être parallèle, et dans ce cas il encapsule une application parallèle dont l'exécution se fait de manière distribuée.

Observation. Dans le canevas d'adaptation dynamique présenté dans [45], la fonction d'observation doit fournir deux services, la notification et l'interrogation. Elle peut surveiller tout type d'informations par le biais d'une interface. Dans Dynaco, le choix a été fait d'externaliser cette fonction.

Décision. Buisson distingue la décision et la planification. La décision choisit la nouvelle configuration à adopter grâce à une politique. La planification définit par quelle moyen l'adopter à l'aide d'un guide. Dans un composant Dynaco, ces fonctions sont implémentées par des composants « décideur » et « planificateur » qui reposent

respectivement sur une politique et un guide. La politique et le guide sont spécifiques à un composant adaptable donné.

Aucune contrainte n'est faite sur le formalisme utilisé pour exprimer la politique. Dans le prototype DYNACO, trois formalismes sont proposés : code JAVA, système expert JESS ou algorithme génétique.

Reconfiguration. La reconfiguration est réalisée par le composant « exécuteur » qui exécute le plan défini par le « planificateur ». L'« exécuteur » repose sur un coordinateur qui choisit à quel moment exécuter le plan. C'est le rôle du coordinateur de garantir la cohérence de la reconfiguration.

Gestion de la cohérence. Buisson donne le modèle théorique suivant. Chaque composant séquentiel qui constitue le composant parallèle détermine ses candidats locaux d'adaptation, qui peut être dans le passé ou dans le futur. Un candidat global est une combinaison des candidats locaux. Pour être valide et garantir la cohérence de la reconfiguration, ce candidat global doit vérifier le *prédicat de cohérence global* [45]. Ce prédicat exprime les conditions nécessaires pour qu'une combinaison de candidats locaux forme un candidat global correct. Le prédicat de cohérence global dépend à la fois de l'application et de la reconfiguration. Les candidats globaux sont donc l'ensemble des combinaisons de candidats locaux qui satisfont le prédicat global de cohérence. Ensuite, le choix d'un point global d'adaptation parmi les candidats globaux se fait selon une métrique définie.

AFPAC [46] est la solution proposée par Buisson pour assurer la cohérence lors de la reconfiguration de composants adaptables parallèles. Cependant AFPAC se limite aux applications SPMD[5]. Dans AFPAC, le prédicat de cohérence global est l'identité. Cela signifie que les candidats globaux d'adaptation valides sont ceux dont tous les composants séquentiels sont au même endroit du calcul. AFPAC propose un algorithme distribué qui permet de trouver le point d'adaptation global et de guider les processus vers ce point.

TACO [45] est l'outil qui permet une annotation automatique de programme MPI. Ceci permet de connaitre son graphe de controle qui sert à déterminer les candidats locaux d'adaptation.

Évaluation. Buisson propose un modèle théorique très général et très complet qui permet d'envisager tout type d'adaptation. Il met en avant la séparation de préoccupations pour bien distinguer le code de l'application et les différents composants intervenant dans l'adaptation. Au final, le processus d'adaptation est complexe et chaque adaptation doit être conçue par un *expert en adaptation* (sic).

Le passage à la pratique est assez limité puisqu'il est restreint aux applications SPMD pour la partie gestion de cohérence de la reconfiguration avec AFPAC et aux programmes MPI pour TACO. En effet, l'absence d'une représentation abstraite du calcul dans le modèle de programmation MPI rend ce problème difficile dans le cas général.

[5] *Single Program, Multiple Data*

3.4.1.3 ASSIST

ASSIST[4, 3, 5, 147] est un environnement d'exécution d'applications distribuées. Les applications sont décrites sous la forme de graphes de modules à l'aide du langage de coordination (*Coordination Language*) ASSIST-CL. Comme le montre la figure 3.3, les modules sont interconnectés par des flux de données typées. Chaque module peut être parallèle ou séquentiel.

Un module parallèle (*parmod*) est utilisé pour décrire l'exécution parallèle d'un certain nombre de fonctions séquentielles. Ils peuvent être de deux types : *Topology none* qui correspond à un modèle maitre-esclaves ; ou *Topology array* qui correspond à un modèle SPMD.

Le langage ASSIST-CL [147] permet de décrire la structure de calcul des modules parallèles tout en faisant appel aux fonctions C, C++ ou FORTRAN pour les parties séquentielles. Cette structure permet de construire un graphe de flot de contrôle du module parallèle lors de la compilation. ASSIST-CL comporte l'instruction `sync` qui agit comme une barrière globale.

a

```
generic subgraph(input_stream long[N] in, long flag
                 output_stream long out) {
  stream long[N] A;
  stream long    B;

  normalize (input_stream in, flag output_stream A);
  parallel  (input_stream A, flag output_stream B);
  filter    (input_stream B output_stream out);
}
```

b

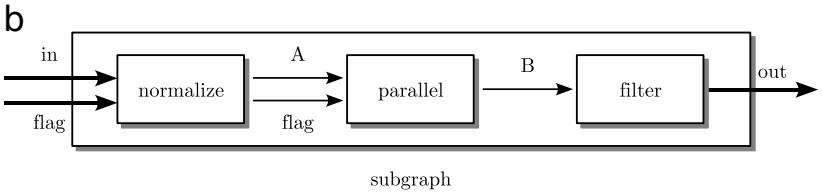


FIG. 3.3: Module `subgraph` avec ASSIST-CL : (a) donne le code source du module ; (b) donne sa représentation sous forme de graphe

Observation. ASSIST utilise la notion de contrat associé au module. Ce contrat spécifie les informations qui doivent être observées et également les objectifs de performance exprimés en fonction des valeurs observées. Lorsque les objectifs ne sont pas atteints, une adaptation est déclenchée.

Reconfiguration. Les reconfigurations proposées par ASSIST pour un module se limitent à l'ajout de nouvelles ressources, à un rééquilibrage du calcul ou éventuellement à la migration entière du module [5].

Décision. Le choix d'une nouvelle reconfiguration dans ASSIST est réalisé en utilisant le modèle de performance fourni avec le contrat du module. Ce modèle de performance est analysé pour déterminer un changement dans la configuration actuelle qui permettra d'améliorer les performances. Cette analyse peut être réalisée facilement puisque les types de reconfigurations sont limités.

Gestion de la cohérence. La gestion de la cohérence d'un module parallèle dans ASSIST est déléguée à une entité appelée *configuration manager*. Le *configuration manager* est responsable de trouver les points sains pour la configuration dans l'exécution du module. ASSIST identifie deux types d'états sains pour la reconfiguration dans son modèle de programmation :

- les barrières globales, qui sont indiquées dans le code du module parallèle par l'instruction `sync` (*on-barrier reconfiguration*) ;
- la fin d'une étape de calcul parallèle, qui signifie que l'état interne distribué est sain et que le module est en attente de données (*on-stream reconfiguration*).

Les concepteurs d'ASSIST soulignent que ce ne sont pas les seuls états sains existants pour la configuration, mais ce sont des points faciles à trouver pour un compilateur et qu'ils suffisent pour les scenarios réels. ASSIST est capable de trouver et de guider l'application vers de tels états (en bloquant l'arrivée des nouvelles données par exemple) [147].

Évaluation. L'utilisation d'ASSIST pour l'adaptation semble assez simple puisque cela nécessite deux étapes : décrire le parallélisme du module parallèle avec ASSIST-CL et définir le contrat d'adaptation. ASSIST a une connaissance de la structure du calcul parallèle de l'application ce qui permet de garantir automatiquement la cohérence des reconfigurations. Cependant, ASSIST est limité aux applications de type maitre-esclaves ou SPMD, et les reconfigurations sont uniquement géométriques ou structurelles.

3.4.1.4 PCL

PCL (*Program Control Language*) [1, 69, 68] est un environnement pour concevoir et implémenter des applications adaptatives distribuées. Il repose sur deux aspects : la construction d'un graphe statique de tâches (*Static Task Graph*) offrant une vue globale de l'application distribuée, et un langage permettant de spécifier les différentes fonctions de l'adaptation.

PCL s'utilise comme un compilateur. La construction du graphe statique de tâches se veut peu intrusive puisqu'elle s'effectue en ajoutant des labels au code source de l'application. L'utilisation de ces labels permet ensuite de désigner dans PCL les tâches ou les parties du code concernées par les différentes parties de l'adaptation. La figure 3.4 montre un aperçu de la structure d'une application adaptative dans PCL.

Observation. La fonction d'observation dans PCL offre les services d'interrogation et de notification par l'intermédiaire des notions de métrique (*Metric*) et d'évènement (*Event*). La métrique définit une valeur qui peut être mesurée, et l'évènement permet de déclencher une adaptation lorsqu'une condition est vérifiée. Une fois définie, la métrique peut être interrogée à distance de manière transparente ; le compilateur génère le code d'appel distant automatiquement.

Décision. Dans PCL, la décision est réalisée par une fonction appelée *Adaptor* qui est écrite dans un langage de programmation classique. Cette fonction peut décider d'effectuer certaines reconfigurations en interrogeant les métriques définies.

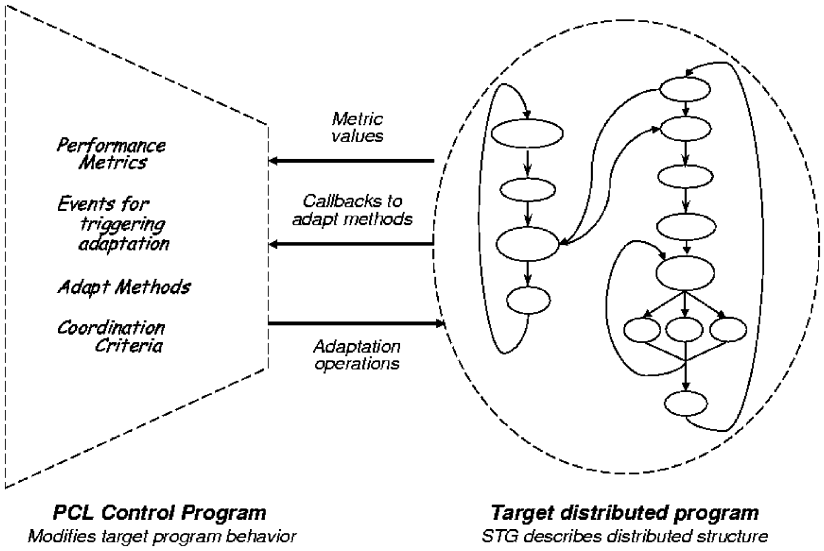


FIG. 3.4: Aperçu de la structure d'une application adaptative dans PCL [69]

Reconfiguration. Les reconfigurations proposées dans PCL sont de deux types. Le premier type correspond aux opérations de modification du graphe statique de tâches qui peuvent être l'ajout, la suppression ou le remplacement d'une tâche, ou bien l'ajout ou la suppression d'un arc. Le deuxième type est le changement de valeur d'un paramètre (qui aura préalablement été identifié dans PCL). Toutes ces reconfigurations peuvent s'appliquer localement ou globalement.

Gestion de la cohérence. La gestion de la cohérence repose en partie sur le programmeur de l'adaptation qui doit spécifier les contraintes de cohérence liées à son adaptation. Pour cela, il doit donner une contrainte interne (*internal constraint*) et une contrainte externe (*external constraint*) [68].

La contrainte interne utilise la notion de région du graphe. Une région correspond à un ensemble de tâches. Une région est dite active si une tâche de la région est en cours d'exécution. La contrainte interne d'une reconfiguration s'exprime par une région et une politique interne. Elle permet d'indiquer quand la reconfiguration peut s'exécuter relativement à cette région. Les choix de politique interne peuvent être :

- `Any` : la reconfiguration peut s'exécuter à tout moment ;
- `RegionIn` : la reconfiguration ne peut s'exécuter qu'à l'entrée de la région spécificiée ;
- `RegionOut` : la reconfiguration ne peut s'exécuter qu'à la sortie de la région spécificiée ;
- `OutsideRegion` : la reconfiguration ne peut s'exécuter que lorsque la région est inactive.

La contrainte externe permet d'exprimer l'état relatif (du point de vue du numéro d'itération) nécessaire entre les processus participant à la reconfiguration.

- `Any` indique qu'il n'y a aucune contrainte entre l'état des processus ;
- `EqualRegionCounters` indique que les processus doivent être au même numéro d'itération.

Ces contraintes permettent à l'utilisateur d'indiquer les conditions nécessaires pour

garantir la cohérence d'une adaptation. Grâce à ces informations, PCL choisit un état sain pour appliquer la reconfiguration.

Évaluation. PCL propose un mécanisme d'adaptation indépendant du modèle de parallélisation. Il offre une représentation abstraite de la structure du calcul d'une manière peu intrusive. Il offre également le support pour réaliser l'observation du contexte d'exécution.

PCL demande un effort de l'utilisateur puisqu'il doit spécifier lui-même la région dans laquelle peut s'exécuter sa reconfiguration, mais cela permet des reconfigurations plus fines car il peut exprimer des reconfigurations locales, globales ou même désynchronisées.

La représentation abstraite du calcul sous forme d'un graphe est au cœur du mécanisme d'adaptation puisque les reconfigurations sont exprimées en termes d'opérations sur ce graphe. Elle permet également à PCL d'assurer la cohérence des reconfigurations grâce aux spécifications données par l'utilisateur.

3.4.1.5 AMPI

AMPI (*Adaptive MPI*) [87, 34] est une implémentation de MPI basée sur Charm++. AMPI propose des fonctionnalités d'adaptation pour les programmes MPI grâce à quelques extensions. L'approche de AMPI est de fournir de l'adaptivité pour MPI par l'intermédiaire de processeurs virtuels. Le programme MPI original est sur-décomposé en V processeurs virtuels pour exhibiter plus de parallélisme, mais ces processeurs virtuels seront exécutés sur seulement P processeurs physiques. Finalement, les V processeurs virtuels sont chacun exécutés dans V processus légers qui sont repliés sur P processus. Pour bénéficier de tous les avantages de AMPI, il est nécessaire de sur-décomposer l'application, c'est-à-dire de prendre V bien plus grand que P.

L'architecture du mécanisme d'équilibrage de charge adaptatif de AMPI est présentée à la figure 3.5.

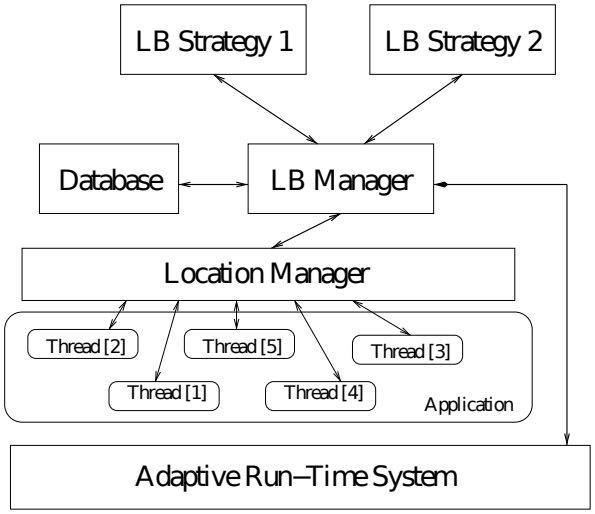


Fig. 3.5: Composants du mécanisme d'équilibrage de charge adaptatif de AMPI [87]

Reconfiguration. Les reconfigurations dans AMPI sont limitées à la migration de processeurs virtuels. Cette migration a pour but de fournir un équilibrage de charge adaptatif.

Observation. La fonction d'observation est réalisée par un composant appelé *Load Balancing Manager*. Il collecte la charge de chaque machine et le temps d'inactivité de chaque processus. Ces informations sont centralisées dans la base de données *Load Balancing Database* [87].

Décision. La fonction de décision est activée par l'appel collectif dans le code de l'application à l'extension AMPI `AMPI_Migrate` ou `AMPI_Async_Migrate`. L'appel à cette fonction ne déclenche pas forcément la migration mais va demander au composant *Load Balancing Manager* de prendre la décision.

Pour prendre sa décision, le *Load Balancing Manager* repose sur un composant de stratégie appelé *Load Balancing Strategy*. Plusieurs stratégies peuvent être implémentées. La stratégie utilisée va déterminer, en utilisant les informations de la base de données *Load Balancing Database*, les processeurs virtuels qui doivent être migrés et à quel moment.

Gestion de la cohérence. Les reconfigurations étant limitées à la migration de processus légers, le problème de la cohérence de la reconfiguration est simplifié. Il repose sur deux aspects. Premièrement, les points de migration sont indiqués dans le code source de l'application par l'intermédiaire des extensions `AMPI_Migrate` ou `AMPI_Async_Migrate` qui représente des barrières dans le calcul. Deuxièmement, le moteur d'exécution AMPI permet de faire suivre les messages d'un processeur virtuel une fois qu'il a migré.

Évaluation. AMPI propose une solution intéressante pour l'adaptation de programme MPI puisqu'elle nécessite très peu de modification de code source de l'application. Les reconfigurations possibles restent limitées à cause du modèle de programmation MPI peu expressif. Seule la sur-décomposition permet d'obtenir une représentation abstraite de l'application sous forme de processeurs virtuels.

Enfin, AMPI donne une faible garantie sur la cohérence de la reconfiguration qui repose entièrement sur le placement des appels aux extensions de AMPI placés par l'utilisateur.

3.4.1.6 Satin

Satin [158, 157] est un environnement de programmation parallèle qui propose des fonctionnalités d'adaptation. Les applications Satin sont basées sur le principe « diviser pour régner ». Les tâches de l'application sont ordonnancées par vol de travail. Une application Satin est donc malléable, c'est-à-dire que des processeurs peuvent joindre ou quitter le calcul en cours d'exécution.

Le mécanisme d'adaptation proposé par Satin repose sur cette hypothèse de malléabilité de l'application. Le but de l'adaptation dans Satin est d'ajouter ou de

supprimer les processus de manière à utiliser les ressources qui fournissent la meilleure efficacité pour l'application.

Observation. L'exécution est divisée en périodes d'observation. Durant une période d'observation, chaque processeur mesure son temps d'inactivité et son temps passé à communiquer et il peut ainsi calculer son surcout d'exécution sur cette période. Ceci permet de calculer les surcouts par processeur, intra-grappe, inter-grappes et globaux. SATIN observe également la vitesse relative des processeurs en mesurant périodiquement sur chaque processeur le temps d'exécution d'un sous-calcul représentatif de l'application. Grâce à toutes ces informations, SATIN peut calculer l'efficacité moyenne pondérée de l'exécution de l'application.

Reconfiguration. Les seules reconfigurations envisagées par SATIN sont l'ajout ou la suppression de processus. Pour cela, le mécanisme d'adaptation suppose que l'application est malléable.

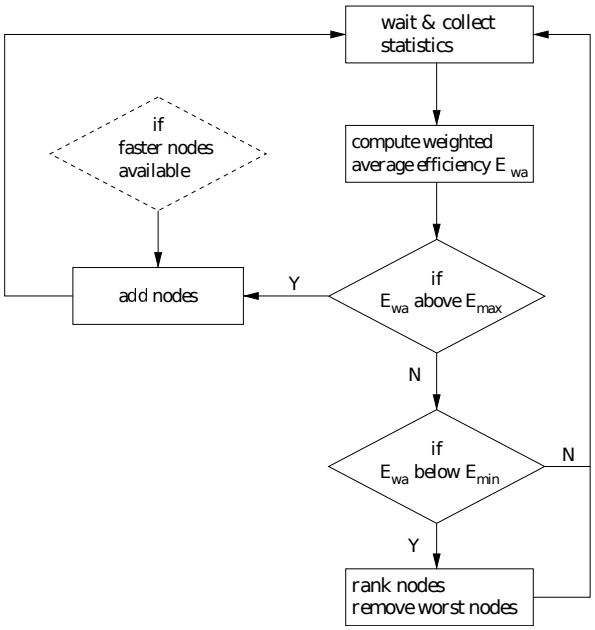


FIG. 3.6: Stratégie de décision dans SATIN [158]

Décision. La fonction de décision de SATIN est donnée à la figure 3.6. Elle est basée sur un algorithme *ad hoc* lié à la fois au critère de performance (l'efficacité de l'exécution) et aux reconfigurations envisagées (ajout ou suppression de processus). L'algorithme de décision est assez simple (*cf* figure 3.6). L'efficacité moyenne pondérée est calculée ; si elle est supérieure au seuil E_{max} alors des processus sont ajoutés ; si elle est inférieure au seuil E_{min} alors les plus mauvais processus sont retirés. Cette heuristique n'offre pas de garantie de résultat mais en pratique elle permet d'obtenir une amélioration des performances pour plusieurs scenarios typiques des grilles de calcul [158].

Gestion de la cohérence. La gestion de la cohérence d'une reconfiguration par SATIN est déléguée à l'application puisque le mécanisme d'adaptation suppose l'application malléable. Cette hypothèse est vérifiée lorsque l'application repose sur le modèle d'exécution de SATIN.

Évaluation. SATIN s'intéresse surtout à la partie décision de l'adaptation et se restreint à la problématique de la sélection de ressources. L'originalité de l'approche proposée par SATIN est que la décision de l'adaptation ne dépend pas d'un modèle de performance de l'application ou d'un ensemble de règles fournies par l'utilisateur. Cependant cet algorithme de décision ne décide que de l'ajout ou de la suppression de ressources.

3.5 Conclusion

Ce chapitre a présenté l'adaptation et la reconfiguration dynamique et a précisé le lien qu'il y a entre ces deux notions.

L'adaptation est un processus qui vise à modifier le système considéré en réponse à un changement dans son contexte d'exécution. L'objectif de l'adaptation est de rendre le système plus performant dans son nouveau contexte. Nous avons donné une classification des travaux portant sur l'adaptation et il en ressort un certain consensus pour distinguer trois fonctions élémentaires qui composent l'adaptation : l'observation, la décision et la reconfiguration.

Nous avons ensuite ciblé notre étude sur la fonction de reconfiguration pour les applications de calcul distribué. La littérature nous a permis de mettre en évidence des critères de classification et les propriétés des reconfigurations. En particulier, la cohérence d'une reconfiguration est un point critique et difficile pour les systèmes distribués. Elle est composée de trois aspects : l'intégrité structurelle, la cohérence mutuelle et les invariants de l'état de l'application.

De point de vue de la conception d'une application distribuée reconfigurable et adaptable, la diversité des solutions récentes proposées indique que le problème est encore ouvert. On peut distinguer d'un côté les solutions qui se limitent à des reconfigurations données (comme par exemple l'ajout et le retrait de machines) et de l'autre, celles portant sur des reconfigurations générales. Ces dernières utilisent une représentation abstraite du futur de l'application (principalement sous forme d'un graphe) pour permettre de déterminer automatiquement un état sain pour la reconfiguration. Cet état sain est appelé point de reconfiguration et permet de garantir la cohérence de la reconfiguration. Certains travaux proposent également au programmeur de spécifier lui-même les contraintes de son opération de reconfiguration.

Ce chapitre nous a permis d'exposer les concepts-clefs indispensables à la mise en place d'un mécanisme de reconfiguration pour un environnement de calcul distribué. La partie II de ce manuscrit porte sur la conception et la mise en œuvre de mécanismes originaux de reconfiguration pour les applications parallèles. Ces mécanismes seront évalués grâce à l'environnement KAAPI.

II

Mécanismes de reconfiguration pour les applications parallèles

Modèle de programmation et moteur d'exécution 4

Sommaire

4.1 Introduction

Ce chapitre présente le modèle de programmation et d'exécution sur lequel repose le mécanisme de reconfiguration dynamique qui sera présenté au chapitre suivant.

De nombreux modèles de programmation ont été proposés pour faciliter la programmation des applications parallèles. Ces modèles proposent différents niveaux d'abstraction. Au plus proche de la machine, on trouve les modèles de programmation comme les processus légers [112] ou MPI [108, 109]. Les processus légers permettent d'exploiter le parallélisme des machines multiprocesseurs ou multicœurs ; MPI permet d'exploiter le parallélisme entre plusieurs machines par échanges de messages.

Parmi les modèles de plus haut niveau, il existe ceux qui sont basés sur le parallélisme de contrôle, par exemple Cilk [39], Intel TBB [99] ou Satin [146] et, ceux qui utilisent le parallélisme de données comme HPF [83] ou BSP [145]. Le modèle flot de données est encore plus général et il est proposé dans Jade [130] et Athapascan [76, 77].

Pour les travaux présentés dans cette thèse, nous avons choisi d'utiliser le modèle de programmation Athapascan et le moteur d'exécution Kaapi. Le moteur d'exécution Kaapi interprète la description de l'application en Athapascan et crée une représentation interne du flot de données pour l'exécuter. Grâce à cette représentation abstraite, le couple Athapascan/Kaapi correspond bien à nos besoins. En effet, comme cela a été

évoqué au chapitre 3, une telle représentation abstraite est très utile pour implémenter des mécanismes d'adaptation et de reconfiguration dynamiques.

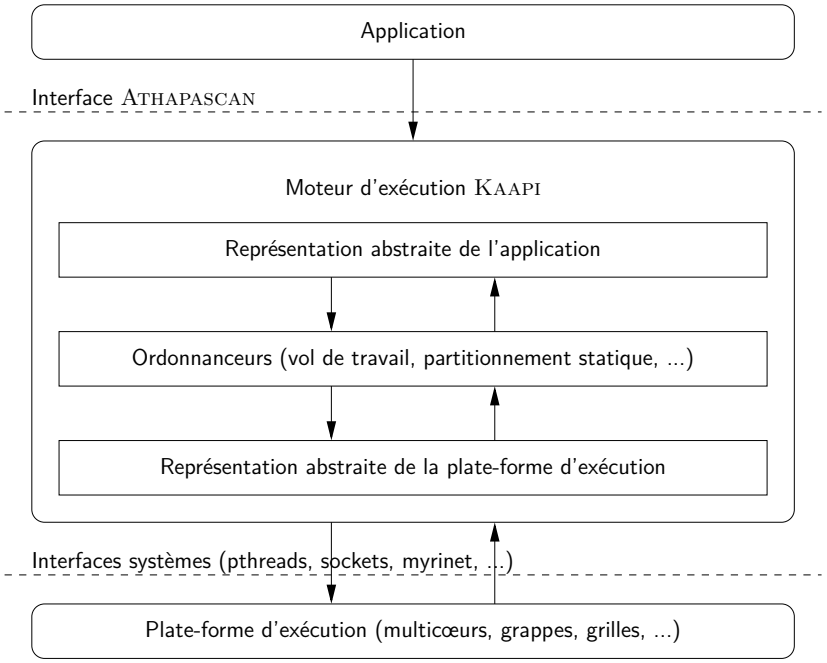


FIG. 4.1: Interface de programmation ATHAPASCAN et moteur d'exécution KAAPI

Comme le montre la figure 4.1, le moteur d'exécution KAAPI est composé de plusieurs couches. La couche la plus basse permet d'abstraire l'architecture matérielle en fournissant une interface indépendante du système et des informations sur l'organisation hiérarchique des cœurs de calcul. La couche la plus haute exécute les instructions du langage ATHAPASCAN pour construire une représentation interne du graphe de flot de données décrit par l'application. La couche intermédiaire interprète ce graphe de flot de données interne et exécute les différentes tâches de calcul de l'application sur les cœurs de calcul disponibles.

Ce chapitre est organisé de la manière suivante. La section 4.2 décrit le modèle de programmation ATHAPASCAN et donne le graphe de flot de données et la sémantique associée. La section 4.3 présente le moteur d'exécution KAAPI et les différentes méthodes d'ordonnancement existantes.

4.2 Modèle de programmation Athapascan

Le modèle de programmation ATHAPASCAN consiste à décrire une application sous forme d'un graphe de flot de données, en reposant sur les notions de **tâches** et de **données partagées**. Les sections suivantes présentent les mots-clés du langage, puis le graphe de flot de données correspondant.

4.2.1 Langage Athapascan

Le langage ATHAPASCAN est une extension du langage C++[1] qui repose sur les mots-clés : `Fork`, `Shared` et `Shared_xx`. Les spécifications, la syntaxe et la sémantique du langage ATHAPASCAN sont détaillées dans [76, 77].

Le mot-clé `Shared<Type>` permet de créer une variable de type `Type` en mémoire partagée, *i.e.* une **donnée partagée**. Une donnée partagée pourra être accédée et modifiée par les tâches de calcul. Les accès des tâches aux données partagées permettent de définir des contraintes sur l'ordre d'exécution des tâches.

Le mot-clé `Fork<Task>()` permet de créer une **tâche** de calcul `Task`. Une tâche représente un appel de fonction C++ de type fonction classe. La création d'une tâche est un appel non bloquant (asynchrone). L'ordre de création des tâches est appelé **ordre de référence**. Lors de la création des tâches, comme en C++, le programmeur peut passer des paramètres par valeurs ou par références (pointeurs). Les paramètres passés **par valeur** correspondent à une copie privée du paramètre au moment de l'appel à `Fork`. Les seuls paramètres passés **par référence** sont les variables en mémoire partagée. Pour ces paramètres, la tâche doit préciser le **droit** (lecture ou écriture) et le **mode d'accès** qu'elle requiert. Cela se traduit par une déclaration d'un paramètre formel avec l'un des types `Shared_<droit/mode>` qui permet de préciser à la fois le droit et le mode d'accès à la donnée en mémoire partagée.

Les droits d'accès actuellement définis dans ATHAPASCAN sont les suivants.

Écriture : qui indique que la tâche va écrire la nouvelle version (paramètre formel de type `Shared_w`) de la donnée ou si la tâche contribue, par accumulation[2], à la future version de la donnée (paramètre formel de type `Shared_cw`).

Lecture : qui indique que la tâche va seulement lire la donnée dans la mémoire partagée (paramètre formel de type `Shared_r`).

Exclusif : qui indique que la tâche prend un accès exclusif à la donnée en lecture et écriture (paramètre formel associé de type `Shared_rw`).

Le droit d'accès que possède une tâche ne peut pas être muté : par exemple, une tâche qui possède un droit en écriture ne peut pas lire la donnée.

Le **mode d'accès** d'une tâche à une donnée partagée permet de définir la portée du droit d'accès. Il peut être direct ou différé.

direct : un mode d'accès **direct** accorde à la tâche elle-même le droit d'accès spécifié. Il permet à la tâche d'accéder elle-même à la donnée partagée en respectant le droit d'accès.

différé : un mode d'accès **différé** accorde le droit d'accès spécifié seulement à la descendance de la tâche. Ainsi un mode d'accès différé ne permet pas à la tâche elle-même d'accéder à la donnée partagée, mais le permet à ses tâches-filles. Un mode d'accès différé est indiqué par la lettre `p` (*postponed*).

[1] Qui est implémenté comme une bibliothèque générique -*template*- en C++

[2] Se référer à [76, 77] : l'opération d'accumulation doit être une fonction classe supposée cumulative et associative pour permettre une évaluation parallèle sans restriction sur un ordre à respecter.

Mot-clé	Type d'accès
`Shared_w`	écriture seule directe
`Shared_wp`	écriture seule différée
`Shared_r`	lecture seule directe
`Shared_rp`	lecture seule différée
`Shared_rw`	modification directe
`Shared_rpwp`	modification différée
`Shared_cw`	écriture cummulée directe
`Shared_cwp`	écriture cummulée différée

TAB. 4.1: Droits et modes d'accès aux données partagées en ATHAPASCAN

Le tableau 4.1 récapitule les modes d'accès d'une tâche à une donnée partagée en ATHAPASCAN. Les règles de compatibilité en les types d'accès ATHAPASCAN sont présentées dans [76, 79].

La figure 4.2 montre un exemple de programme écrit en ATHAPASCAN et le compare à la version séquentielle. Ce programme calcule le $\mathtt{n}^{e}$ terme de la suite de Fibonacci en utilisant l'algorithme récursif naïf. À gauche, nous donnons la version séquentielle écrite en utilisant des fonctions pour définir chaque opération élémentaire. À droite, nous donnons la version ATHAPASCAN qui définit pour chaque opération élémentaire une tâche sous la forme `struct Task::operator()`.

Les deux programmes de la figure 4.2 calculent la même valeur. Les mots-clés d'ATHAPASCAN permettent d'exprimer le parallélisme du calcul en créant explicitement des tâches accédant à des variables en mémoire partagée. Sur l'exemple de la figure (b) aux lignes 21 et 22, la tâche `Fibo` crée deux tâches `Fibo` chargées de calculer les nombres de Fibonacci $n-1$ et $n-2$: ces deux tâches sont indépendantes car elles produisent des données (droit d'accès en écriture) et travaillent sur des variables distinctes de la mémoire partagée. Ensuite à la ligne 23, une tâche `Sum` est créée : le droit d'accès en lecture requis pour ses 2^{e} et 3^{e} paramètres fera que le moteur d'exécution KAAPI ne l'exécutera que lorsque ses paramètres effectifs `res1` et `res2` auront été produits à la fin de l'exécution des tâches `Fibo` précédemment créées.

Le parallélisme d'une application ATHAPASCAN est explicite (c'est au rôle du programmeur de décrire les tâches et les variables en mémoire partagée) mais il est indépendant de l'architecture. Le moteur d'exécution KAAPI, et en particulier l'algorithme d'ordonnancement, décidera ou non d'utiliser le parallélisme de l'application selon les ressources de calcul disponibles.

Néanmoins, tel quel ATHAPASCAN n'est pas directement utilisable pour décrire des applications dans lesquelles le degré de parallélisme doit être proche du nombre de ressources utilisées : en effet, si la gestion du parallélisme implique l'ajout d'un grand nombre d'opérations arithmétiques, il vaut mieux utiliser un algorithme d'extraction adaptatif pour éviter la perte d'efficacité [140]. De même ATHAPASCAN n'est pas bien adapté pour la description des schémas itératifs pour lesquels il est préférable d'ordonnancer l'ensemble des tâches du corps de boucle. Ce dernier point est l'objectif de la section suivante.

```
1  void Sum(
2
3    int* res,
4    int a,
5    int b )
6  {
7    *res = a + b;
8  }
9
10
11 void Fibo (
12
13  int* res,
14  int n )
15 {
16   if ( n < 2 )
17     *res = n;
18   else
19   {
20     int res1, res2;
21     res1 = Fibo( n-1 );
22     res2 = Fibo( n-2 );
23     Sum( res, res1, res2 );
24   }
25 }
26
27
28 void fibonacci( unsigned int n )
29 {
30   int res;
31   res = Fibo( n );
32   Print( res );
33 }
```

(a) Programme séquentiel

```
1  struct Sum {
2    void operator() (
3      a1::Shared_w<int> res,
4      a1::Shared_r<int> a,
5      a1::Shared_r<int> b)
6    {
7      res.write( a.read() + b.read() );
8    }
9  };
10
11 struct Fibo {
12   void operator() (
13     a1::Shared_w<int> res,
14     int n )
15   {
16     if ( n < 2 )
17       res.write( n );
18     else
19     {
20       a1::Shared<int> res1, res2;
21       a1::Fork<Fibo>()(res1, n-1);
22       a1::Fork<Fibo>()(res2, n-2);
23       a1::Fork<Sum>() (res, res1, res2);
24     }
25   }
26 };
27
28 void fibonacci( unsigned int n )
29 {
30   a1::Shared<int> res;
31   a1::Fork<Fibo>()  ( res, n );
32   a1::Fork<Print>() ( res );
33 }
```

(b) Programme ATHAPASCAN

FIG. 4.2: Programmes récursifs qui calculent la suite de Fibonacci

4.2.2 Extension du langage Athapascan

Cette section propose une extension du langage ATHAPASCAN pour les applications itératives comme celles considérées dans [117]. Cette extension repose sur le mot-clé **While** qui permet d'exprimer une boucle de calcul et l'arrêt sur une condition.

Trois éléments sont nécessaires pour définir cette boucle en ATHAPASCAN. Ces trois éléments apparaissent dans l'utilisation de l'instruction **While** que nous avons introduite dans le cadre de ce travail.

- Un donnée partagée contenant la valeur de la condition de répétition qui sert à arrêter la boucle. Tant que la valeur de cette donnée partagée est vraie, la boucle est répétée ; lorsqu'elle devient fausse, la boucle s'arrête.
- La tâche représentant le noyau de calcul (*kernel*) de la boucle est la partie du programme qui doit être répétée tant que la condition de répétition est vraie.
- La tâche représentant le calcul de la condition de répétition est la partie du programme qui permet de calculer la valeur de la condition de répétition.

L'exécution de l'instruction

```
a1::While<Kernel,Test> ( condition )
    ( /* parametres Kernel */ )
```

```
( /* extra parametres Test */ );
```

correspond à la création d'une tâche spéciale appelée **Loop**. L'exécution des itérations est réalisée de manière asynchrone de la même manière que pour le **Fork**. Les contraintes induites par les types d'accès sont les mêmes.

La sémantique de l'instruction **While** est définie pour être équivalente à l'exécution du programme séquentiel C++ suivant :

```
while( condition )
{
  Kernel() ( [ parametres Kernel] );
  Test()   ( condition [, extra parametres Test ] );
}
```

L'exécution de la tâche **Loop** sera équivalente à l'exécution des tâches **Kernel** et **Test** tant que la valeur de la donnée partagée `condition` est vraie. La tâche **Test** doit modifier la donnée partagée `condition` pour permettre l'arrêt de la boucle.

Séparer le noyau de calcul et le test de répétition permet de dérouler le corps de la boucle indépendamment du test d'arrêt. Typiquement, si on suppose que le noyau de calcul fait décroitre une valeur[3] (par exemple une erreur) et que la condition d'arrêt est le franchissement d'un seuil, il peut être utile de réaliser un contrôle amorti du test d'arrêt en ne testant pas la condition à chaque itération [22][4].

Un second intérêt de cette séparation est d'identifier facilement ces tâches afin de ne pas les évaluer lors de la reprise après panne : pour toutes les itérations perdues nous savons que ces tâches ont retourné la valeur *vraie*, sinon le programme se serait arrêté en cours d'itération.

Pour illustrer une telle boucle de calcul, nous utilisons un exemple de programme qui calcule la $\mathtt{n}^{\mathrm{e}}$ ligne du triangle de Pascal. Le triangle de Pascal est une représentation géométrique des cœfficients binomiaux. À la ligne i et à la colonne j $(0 \leq j \leq i)$, on trouve le cœfficent binômial $\binom{i}{j}$. La figure 4.3 montre l'affichage attendu pour un appel à la fonction `pascal`.

```
pascal(0)  →  1
pascal(1)  →  1  1
pascal(2)  →  1  2  1
pascal(3)  →  1  3  3  1
    ⋮         ⋮              ⋱
pascal(n)  →  1  n  ...  ...  n  1
```

FIG. 4.3: La fonction `pascal` calcule la $\mathtt{n}^{\mathrm{e}}$ ligne du triangle de Pascal

La figure 4.4 montre deux programmes réalisant de manière itérative la fonction `pascal` définie ci-dessus. À gauche (figure (a)), la version séquentielle a été écrite pour exprimer chaque opération élémentaire dans une fonction indépendante. De plus, la

[3] Il faut supposer cette valeur décroissante monotone.

[4] Cette possibilité peut être indiquée par le programmeur en positionnant l'attribut `relaxed` sur la tâche **Loop**.

boucle de calcul de la fonction `pascal` a été écrite pour décrire la sémantique de la boucle de calcul du programme ATHAPASCAN présenté à droite (figure (b)).

```
1
2
3  void Add( int& a, int& b )
4  {
5    a += b;
6  }
7
8
9
10
11 bool Test(
12     int a,
13     int b )
14 {
15   return a < b;
16 }
17
18
19
20
21 void Kernel( std::vector<int>& tab,
22               int& curr_iter )
23 {
24   int i;
25   for (i = tab.size()-1; i > 0; i--)
26     Add(tab[i], tab[i-1]);
27   curr_iter++;
28 }
29
30
31 void pascal( int n )
32 {
33   std::vector<int> tab;
34   tab.resize( n );
35
36   ... // Initialisation de tab
37
38   bool condition = true;
39   int curr_iter = 0;
40   while( condition )
41   {
42     Kernel( tab, curr_iter );
43     condition = Test( curr_iter, n )
44   }
45
46   ... // Affichage de tab
47 }
```

(a) Programme séquentiel

```
1  struct Add {
2    void operator() ( a1::Shared_rw<int> a,
3                      a1::Shared_r<int> b )
4    {
5      a.access += b.read();
6    }
7  };
8
9  struct Test {
10   void operator() (
11       a1::Shared_w<bool> condition,
12       a1::Shared_r<int> a,
13       int b )
14   {
15     condition.write( a.read() < b );
16   }
17 };
18
19 struct Kernel {
20   void operator() (
21     std::vector<a1::Shared_rpwp<int> >&tab,
22     a1::Shared_rw<int> curr_iter )
23   {
24     int i;
25     for (i= tab.size()-1; i > 0; i--)
26       a1::Fork<Add>()(tab[i], tab[i-1]);
27     curr_iter.access()++;
28   }
29 };
30
31 void pascal( int n )
32 {
33   std::vector< a1::Shared<int> > tab;
34   tab.resize( n);
35
36   ... // Initialisation de tab
37
38   a1::Shared<bool> condition(true);
39   a1::Shared<int> curr_iter(0);
40
41   a1::While<Kernel,Test> ( condition )
42     ( tab, curr_iter, largeur ),
43     ( condition, curr_iter, nb_iter );
44
45
46   ... // Affichage de tab
47 }
```

(b) Programme ATHAPASCAN

FIG. 4.4: Programmes itératifs qui calculent la n^e ligne du triangle de Pascal

Dans le programme ATHAPASCAN de la figure 4.4a, la boucle d'itération est exprimée à l'aide de l'instruction **`While`**. Cette instruction utilise les trois éléments suivants :

- la donnée partagée `condition` qui contient la valeur de la condition de répétition ;
- une tâche **`Kernel`** qui contient le noyau de calcul à répéter ;
- et la tâche **`Test`** qui calcule la valeur de la condition.

4.2.3 Graphe de flot de données

L'exécution d'une application écrite à l'aide du langage ATHAPASCAN peut être représentée sous forme d'un **graphe de flot de données**. Ce graphe permet de représenter les tâches, les versions des données et les accès des tâches sur les versions.

Nous définissons le graphe de flot de données de la manière suivante :

Définition 5 *Le **graphe de flot de données** associé à l'exécution d'une application* ATHAPASCAN *est un graphe orienté acyclique $G = (S, A)$.*

- *Les tâches S_T et les versions S_V forment l'ensemble des sommets du graphe $S = S_T \cup S_V$.*
- *Les accès des tâches aux données forment l'ensemble A des arcs du graphe.*

Ce graphe est un graphe biparti : un sommet tâche *est uniquement connecté à des sommets* versions *; un sommet* version *est uniquement connecté à des sommets* tâches.

Un arc dans le graphe signifie un accès de type lecture ou écriture entre une donnée et une tâche. Soient $t \in S_T$ une tâche et $v \in S_V$ une version, alors :

- *l'arc $(t, v) \in A$ signifie que la tâche t écrit la version v ; la tâche t précède toute tâche qui lit la version v.*
- *l'arc $(v, t) \in A$ signifie que la tâche t lit la version v ; la tâche t est précédée par toute tâche qui écrit la version v.*

On remarque que si plusieurs tâches écrivent la même version, alors l'accès est un accès en écriture cumulée. De même, si une tâche lit une version et écrit une autre version de la même donnée, alors l'accès est une modification.

Ce graphe est **dynamique** : les sommets *tâches* et *versions* sont ajoutés et supprimés au cours de l'exécution de l'application. Par exemple, la création d'une donnée partagée avec le mot-clé `Shared` ajoute un sommet *version* qui correspond à la version initiale ; la création d'une tâche avec le mot-clé `Fork` ajoute un sommet *tâche* ainsi que les arcs correspondant aux accès directs de la tâche ; un accès en écriture directe sur une donnée partagée crée un nouveau sommet *version*. La construction de ce graphe en cours d'exécution de l'application est détaillée dans [76].

L'aspect dynamique de ce graphe permet de décrire des applications récursives. L'exécution d'une tâche crée les sous-tâches qui la composent et permet d'affiner localement la description du parallélisme.

La figure 4.5 montre les graphes de flot de données associés à l'exécution du programme récursif de calcul de Fibonacci. Le programme ATHAPASCAN correspondant est donné à la figure 4.2b. Ce programme est récursif. Les trois sous-figures (a), (b) et (c), montrent le développement du graphe lors de l'exécution des tâches :

- le graphe de flot de données (a) résulte de l'exécution de la fonction `fibonacci` ;
- le graphe de flot de données (b) correspond au développement du sous-graphe de la tâche `Fibo(3)` ;
- le graphe de flot de données (c) correspond au développement du sous-graphe de la tâche `Fibo(2)`.

Le graphe de flot initial comprend déjà toutes les informations nécessaires à l'exécution, mais le parallélisme interne à une tâche n'apparait que lorsqu'elle est exécutée.

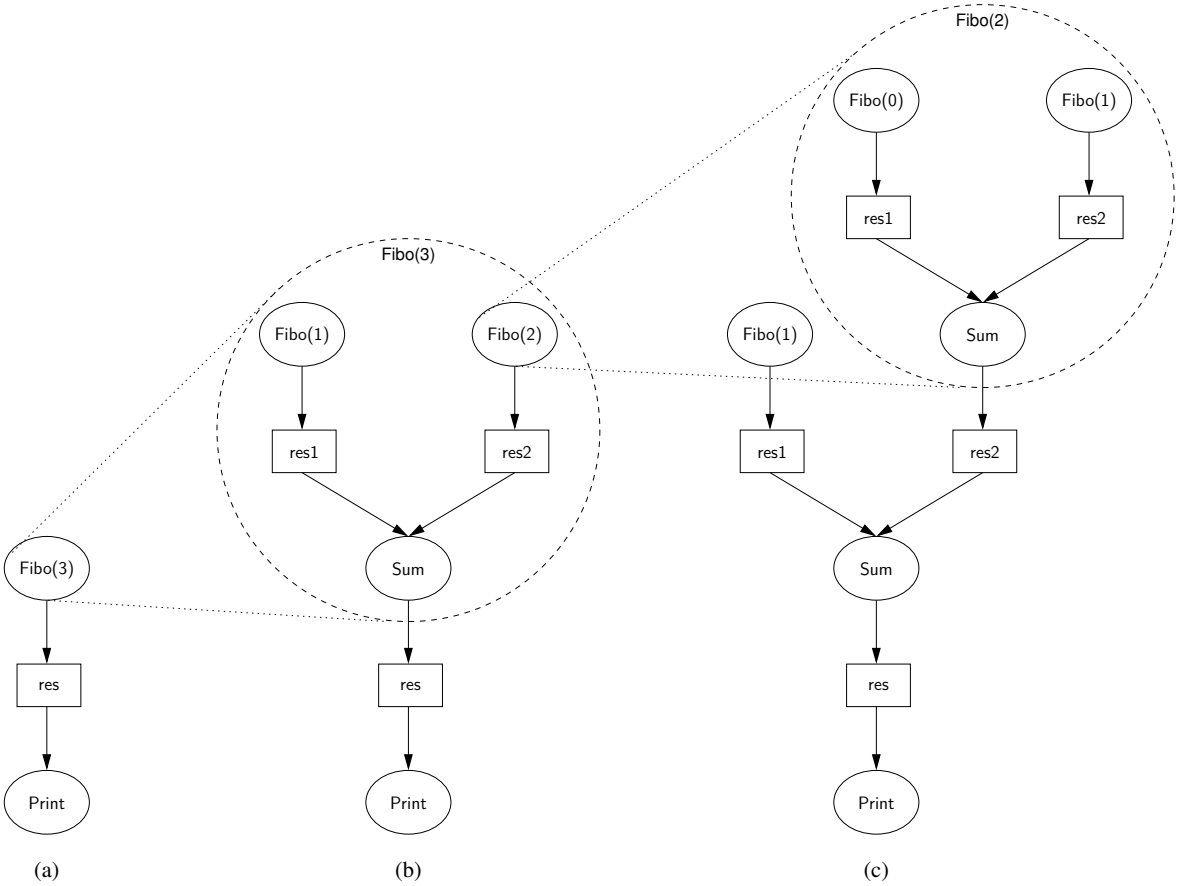


FIG. 4.5: Graphes de flot de données associés au programme ATHAPASCAN de calcul de la suite de Fibonacci de la figure 4.2b

Le cas de programmes ATHAPASCAN itératifs est similaire. La boucle d'itération est représentée par une tâche spéciale **`Loop`**. L'exécution de cette tâche dépend alors de la valeur de la donnée partagée `condition` :

- Si `condition` vaut faux, alors la tâche **`Loop`** ne fait rien ;
- Si `condition` vaut vrai, alors la tâche **`Loop`** ajoute dans le graphe les tâches **`Kernel`** et **`Test`** représentant une itération et insère une nouvelle tâche **`Loop`** représentant les autres itérations.

La figure 4.6 montre les graphes de flot de données associés à l'exécution du programme itératif de calcul du triangle de Pascal. Ce programme est donné à la figure 4.4b. Ce graphe de flot de données est développé à l'exécution lors de l'exécution des tâches :

- le graphe (a) montre la tâche **`Loop`** avec ses accès sur les données ;
- le graphe (b) montre le graphe résultant du développement d'une itération de la boucle ;
- le graphe (c) correspond au graphe développé après l'exécution de la tâche **`Kernel`** et fait apparaitre le calcul interne de la boucle.

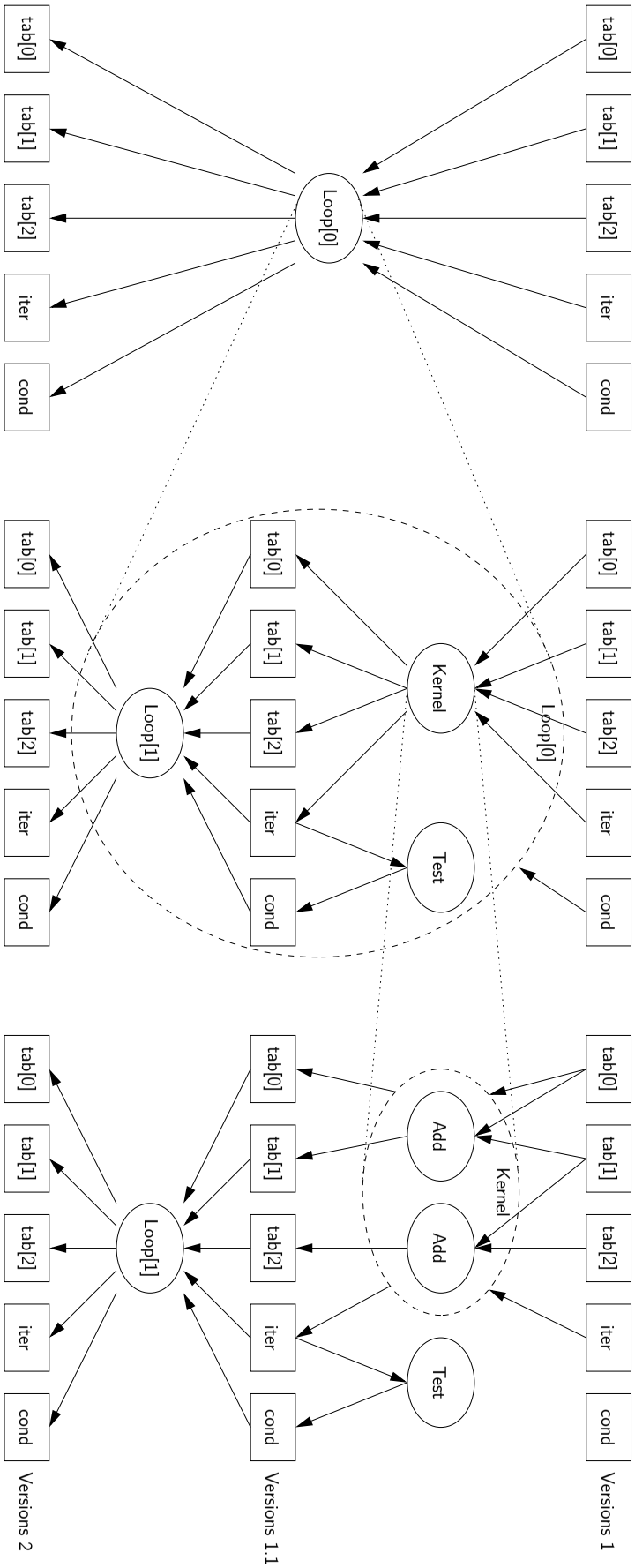


FIG. 4.6: Graphes de flot de données associés à l'exemple Pascal de la figure 4.4b

4.2.4 Sémantique Athapascan

La sémantique du langage ATHAPASCAN est définie par l'ordre séquentiel d'exécution des programmes ATHAPASCAN : ainsi le résultat de l'exécution d'un programme ATHAPASCAN est le même que celui de l'exécution du programme dans lequel les mots-clés du langage ont été retirés.

Plus précisément, l'**exécution séquentielle** d'un programme ATHAPASCAN est définie de la manière suivante :

- une déclaration `Shared` est remplacée par la déclaration d'une variable suivie de son initialisation ;
- un `Fork` est remplacé par l'appel direct de la méthode de calcul ;
- les types d'accès `Shared_xx` sont remplacés par le passage des paramètres par référence C++ (qui permet la lecture et l'écriture) ;
- les accès aux données (`read()`,`write()`,`access()`) sont remplacés par des accès directs ;
- l'instruction `While` est remplacée par une boucle `while` (*cf* section 4.2.2).

Le graphe de flot de données de l'application ATHAPASCAN définit les **contraintes de précédence** entre les tâches. Ces contraintes de précédence définissent un ordre partiel $\prec_G$ qui doit être respecté par toute exécution valide calculée par un ordonnancement.

Définition 6 *Soit G le graphe de flot de données de l'instance*[5] *de l'application considérée. Un ordre d'exécution $\prec$ des tâches de l'application est dit **valide** s'il respecte les contraintes de précédence du graphe de flot de données G.*

*Plus formellement, on dira que l'ordre $\prec$ est **valide** s'il est **compatible** avec l'ordre $\prec_G$, c'est-à-dire que pour tout couple d'éléments (a, b), $(a \prec_G b) \Rightarrow (a \prec b)$.*

Un ordre d'exécution particulier est l'ordre de création des tâches, appelé l'**ordre de référence**. Cet ordre est différent de l'ordre séquentiel car selon cet ordre les tâches-filles sont exécutées après la tâche-mère alors que pour l'ordre séquentiel, les tâches-filles sont exécutées durant l'exécution de la tâche-mère. Dans [76], Galilée montre que le l'ordre de référence est toujours un ordre d'exécution valide.

Dans [76], Galilée définit la **sémantique d'Athapascan** à partir du graphe de flot de données construit par l'interprétation des mots-clés du langage ATHAPASCAN. À partir de cette définition, Galilée prouve que la sémantique d'ATHAPASCAN est telle que pour toute exécution (éventuellement parallèle), la valeur retournée lors d'un accès en lecture à une donnée partagée est identique à la valeur lue lors de l'exécution séquentielle de l'application [76].

Proposition 2 *[76] Quel que soit un ordre d'exécution $\prec$ des tâches **valide**, la valeur retournée lors d'un accès en lecture à une donnée partagée est identique à la valeur lue lors de l'exécution séquentielle de l'application.*

Cette propriété, montrée par Galilée [76], repose sur le fait que les tâches ne modifient pas la mémoire par un effet de bord, c'est-à-dire qu'une tâche ne peut pas accéder, soit

[5] L'instance d'une application est l'association d'une application et d'un ensemble de ses valeurs d'entrée.

directement soit via sa descendance, à un objet pour lequel elle n'a pas déclaré l'accès avec les droits correspondants. Ceci interdit en particulier l'utilisation de variables globales dans les tâches de l'application.

Définition 7 *Étant donné un ordre d'exécution $\prec$ nous noterons par $Valeur_{\prec}(d)$ la valeur calculée du sommet version $d \in S_V$ en utilisant l'ordre $\prec$.*

Galilée a montré que si $\prec_1, \prec_2$ sont deux ordres valides, alors les fonctions valeurs sont identiques, i.e. $Valeur_{\prec_1} \equiv Valeur_{\prec_2}$.

Néanmoins dans le cadre de cette thèse, nous verrons dans les chapitres à suivre comment transformer dynamiquement l'application représentée par son graphe de flot de données. Parmi l'ensemble de transformation à priori possible, certaines préservent la sémantique de l'application et d'autres non. Nous nous intéressons essentiellement aux transformations qui préservent la structure du graphe et ne modifient pas les valeurs calculées (quitte à ajouter des tâches).

Définition 8 *Soit G un graphe de flot de données de l'application à un instant t. Une fonction f qui transforme G en G' est appelée* ***transformation valide*** *si $\prec_{G'}$ est un ordre valide, i.e., $\forall (a,b)$ $a \prec_G b \Rightarrow f(a) \prec_{G'} f(b)$ et pour tout nœud $d \in S_V$ alors $d \in G'$ et $f(valeur(d)) = valeur(d)$.*

Ces transformations servent à caractériser d'une part un ordre d'exécution et d'autre part le fait que les valeurs calculées restent identiques avant et après transformation : elles préservent les nœuds versions du graphe original G' ainsi que l'ordre initial d'exécution des tâches.

4.3 Moteur d'exécution Kaapi

Kaapi est un intergiciel qui permet d'exécuter des applications programmées avec le langage Athapascan sur une architecture distribuée. Comme le montre la figure 4.1, le moteur d'exécution Kaapi se compose de trois parties :

- La représentation abstraite de l'application constitue une structure de données qui représente l'état de l'application sous la forme de graphe de flot de données Athapascan. Ces graphes sont encapsulés dans des objets appelés **K-threads** (comme Kaapi-threads) qui représentent le travail à effectuer. Cette représentation abstraite est présentée dans la section 4.3.1.
- La représentation abstraite de la plate-forme d'exécution constitue une structure de données qui décrit la hiérarchie des processus de calculs. Les cœurs de calcul sont représentés sous la forme d'objets appelés **K-processeurs** (comme Kaapi-processeurs).
- La partie ordonnancement constitue le moteur d'exécution qui, grâce aux informations sur l'état de l'application et l'organisation hiérarchique des cœurs de calcul, effectue les choix d'ordonnancement. Pour cela, le moteur d'exécution découpe les K-threads pour extraire le parallélisme et les place sur les K-processeurs pour les exécuter. Le moteur d'exécution séquentiel et les algorithmes d'extraction du parallélisme sont présentés dans la section 4.3.2.

4.3.1 État de l'application

Le graphe de flot de données présenté à la section 4.2.3 permet de représenter les calculs de l'application sous la forme de tâches. KAAPI utilise ce graphe pour connaitre les calculs à effectuer. Chaque graphe est stocké dans un objet appelé K-thread.

En interne, le graphe de flot de données est annoté à l'aide d'**attributs** qui permettent d'avoir toutes les informations nécessaires à l'exécution. Par exemple, chaque tâche a un attribut qui correspond à son état et un attribut qui identifie le code d'exécution de la tâche. L'annotation du graphe permet de séparer l'état de l'application et le code d'ordonnancement.

Ce graphe de flot de données annoté par les attributs constitue une représentation abstraite du calcul. Lorsqu'aucune tâche n'est en cours d'exécution, le graphe de flot de données complet, c'est-à-dire l'ensemble de tous les K-threads, respecte les conditions nécessaires et suffisantes pour constituer une représentation valide de l'état de l'application [92] :

- Le graphe de flot de données est une représentation **close**, *i.e.* il contient toutes les informations nécessaires pour représenter l'état de l'application. En effet, lorsqu'aucune tâche n'est en cours d'exécution, le graphe de flot de données contient toutes les informations nécessaires à la poursuite du calcul (la valeur des données, les tâches à exécuter et les dépendances) puisque les tâches interfèrent uniquement avec les données passées en paramètres (les effets de bord sur la mémoire sont interdits).
- Le graphe de flot de données est une représentation **causalement connectée** à l'application, *i.e.* toute modification de l'état de l'application est visible dans la représentation et inversement, toute modification de la représentation est répercutée sur l'état de l'application. Ceci est dû au fait que le moteur d'exécution KAAPI interprète le graphe de flot de données pour exécuter l'application. Réciproquement les tâches exécutées et les données modifiées sont directement reportées dans le graphe.

Il est important de remarquer que lorqu'une tâche est en cours d'exécution, le graphe de flot de données n'est plus une représentation valide de l'état de l'application. En effet, dans ce cas l'état de l'application dépend de la position du pointeur d'instructions dans la tâche courante et cette information n'est pas inscrite dans la représentation abstraite.

Une propriété importante de cette représentation abstraite est qu'elle est indépendante de l'architecture de la machine d'exécution[6] (système d'exploitation, processeur) et donc supporte l'hétérogénéité. Toutes ces propriétés permettent au moteur d'exécution KAAPI d'utiliser cette représentation abstraite pour, à la fois manipuler l'état de l'application, mais aussi le sauvegarder et le migrer.

4.3.2 Exécution et ordonnancement

Dans le moteur d'exécution KAAPI, les K-threads représentent le calcul et les K-processeurs représentent les ressources de calcul. L'exécution est réalisée en assignant les K-threads aux K-processeurs.

[6] À la condition que ce graphe et ses attributs soit décrits d'une manière portable.

Initialement, le graphe de flot de données de l'application est contenu dans un seul K-thread. Pour réaliser une exécution parallèle, il est donc nécessaire de découper les K-threads en extrayant le parallélisme exprimé dans le graphe de flot de données. Les K-threads peuvent ensuite être distribués sur les K-processeurs libres. Ce travail d'extraction de parallélisme et de répartition du travail est le rôle de l'ordonnanceur.

La section 4.3.2.1 présente le fonctionnement d'un K-processeur qui réalise une exécution séquentielle. Les sections 4.3.2.2 et 4.3.2.2 présentent respectivement les méthodes d'exécution par vol de travail et par partitionnement statique.

4.3.2.1 Exécution séquentielle

L'exécution séquentielle est la fonction de base du moteur d'exécution Kaapi. Elle est réalisée par un K-processeur qui parcourt les tâches composant le K-thread. L'exécution séquentielle est basée sur l'état des tâches.

Une tâche peut avoir les états suivants :

- Créée : la tâche est créée mais ses contraintes de précédence ne sont pas vérifiées.
- Attente : la tâche est créée mais ne peut pas être exécutée, elle est bloquée. Cet état permet d'empêcher l'exécution d'une tâche même si ses contraintes de précédence sont résolues, en attendant qu'une certaine condition soit vérifiée. Pour s'exécuter, la tâche devra être mise dans l'état Créée lorsque cette condition sera vérifiée. Cet état est par exemple utilisée lorsqu'une tâche doit attendre la réception d'une donnée pour s'exécuter.
- Prête : la tâche peut être exécutée car elle n'est pas bloquée et ses contraintes de précédence sont vérifiées.
- Exécution : la tâche est en cours d'exécution, *i.e.* le code de la fonction associé à la tâche est en train d'être exécuté.
- Volée : la tâche a été volée, elle est bloquée. Cet état est utilisé par l'algorithme de vol de travail pour créer une synchronisation pendant que la tâche est exécutée sur un autre K-processeur.
- Terminée : l'exécution de la tâche est terminée. Elle est maintenant ignorée en attendant sa destruction.

La figure 4.7 donne le diagramme d'état d'une tâche dans le moteur d'exécution Kaapi.

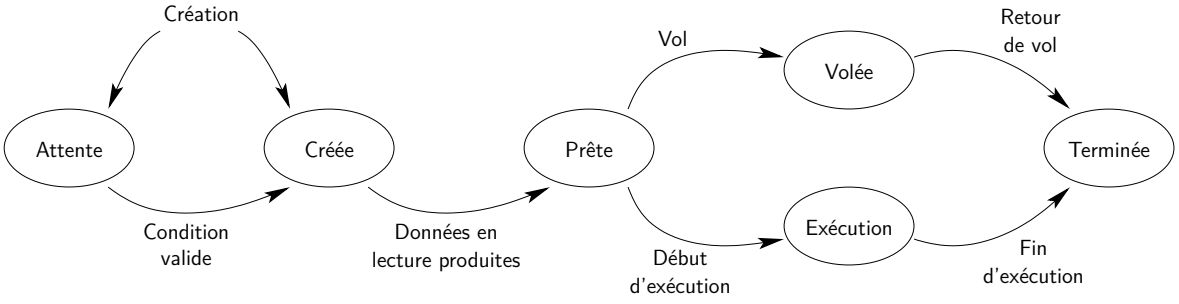


Fig. 4.7: Diagramme d'état d'une tâche dans Kaapi

C'est le K-processeur qui réalise l'exécution des K-threads. Chaque K-thread est associé à un unique K-processeur qui est responsable de son exécution. Le K-processeur

ne peut exécuter qu'un seul K-thread à la fois, donc il doit sélectionner un K-thread parmi les K-threads qui lui sont associés pour l'exécuter. Parmi les K-threads associés à un K-processeur, on distingue les trois ensembles suivants :

- L'ensemble des K-threads prêts, qui sont ceux dont la première tâche est prête.
- L'ensemble des K-threads bloqués, qui sont ceux dont la première tâche est bloquée, c'est-à-dire dans l'état Volée ou Attente.
- Le K-thread actif, qui est le K-thread en cours d'exécution.

L'exécution d'un K-thread est séquentielle et elle se fait selon l'ordre de référence. En effet, comme expliqué dans la section 4.2.4, l'ordre de référence est toujours un ordre valide. Donc pour exécuter les tâches selon cet ordre, il n'y a pas besoin de vérifier les contraintes de précédence. Cela permet une exécution séquentielle avec un faible surcout.

Lorsqu'un K-processeur n'a plus de travail, c'est-à-dire qu'il n'a plus de K-thread prêt, il devient inactif. Cet évènement est traité par l'ordonnanceur qui essaie alors de créer de nouveaux K-threads en extrayant du parallélisme, soit à partir des K-threads bloqués sur le même K-processeur, soit à partir des K-threads des autres K-processeurs (éventuellement distants).

4.3.2.2 Vol de travail

L'ordonnancement par vol de travail est un ordonnancement dynamique de type glouton, c'est-à-dire que s'il existe une tâche prête qui n'est pas en cours d'exécution, alors cela signifie que tous les K-processeurs sont actifs. Une opération de vol de travail s'effectue en plusieurs étapes :

1. Lorsqu'un K-processeur devient inactif, l'ordonnanceur déclenche une opération de vol de travail pour lui fournir des tâches à exécuter. Ce K-processeur est appelé le **K-processeur voleur**.
2. L'algorithme de vol de travail choisit un K-thread dans lequel il va tenter de trouver une tâche à voler. Ce K-thread est appelé le **K-thread victime**. Plusieurs politiques existent pour choisir la victime d'un vol ; la plus courante est le choix aléatoire [39].
3. L'algorithme de vol de travail parcourt le K-thread victime à la recherche d'une tâche qui peut être volée, c'est-à-dire une tâche prête. Pour cela, l'algorithme de vol de travail parcourt et analyse le graphe de flot de données. La tâche sélectionnée est appelée la **tâche volée** et son état est mis à Volée.
4. Un nouveau K-thread est créé. Il contient une copie de la tâche volée et de ses données d'entrée ; et également une tâche spéciale, appelée `Signal`, qui dépend des données écrites par la tâche volée.
5. Ce nouveau K-thread est finalement associé au K-processeur voleur.

Le rôle de cette tâche spéciale `Signal` est d'effectuer ce qui est appelé le **retour de vol**. Cela consiste à retourner au K-thread victime les résultats produits par la copie de la tâche volée (*i.e.* les données en écriture), puis à débloquer la tâche volée sur le K-thread victime en mettant son état à Terminée car les données ont été écrites.

Proposition 3 *L'opération de vol de travail est une* transformation valide.

Preuve Nous donnons l'idée de la démonstration.

- Au moment du vol, la tâche volée est prête, ce qui signifie que tous ses prédécesseurs ont été exécutés. Donc la copie de la tâche volée s'exécute bien après tous les prédécesseurs de la tâche volée.
- Dans le nouveau K-thread, la tâche `Signal` dépend des données produites par la copie de la tâche volée ; elle s'exécute donc après la copie de la tâche volée. Les données lues et émises sont donc correctes.
- Dans le K-thread victime, la tâche volée est dans l'état Volée tant que la tâche `Signal` n'a pas écrit les données produites par la copie de la tâche volée. Donc la tâche volée ne peut pas être exécutée ; et donc toutes les autres tâches qui utilisent ses résultats ne peuvent s'exécuter.
- Dans le K-thread victime, lorsque la tâche volée passe dans l'état Terminée, les données ont été écrites. Les successeurs de la tâche volée peuvent s'exécuter et utilisent les valeurs correctes des données.

Les contraintes d'ordre du graphe de flot de données initial sont respectées et les valeurs des versions des données sont reportées correctement. □

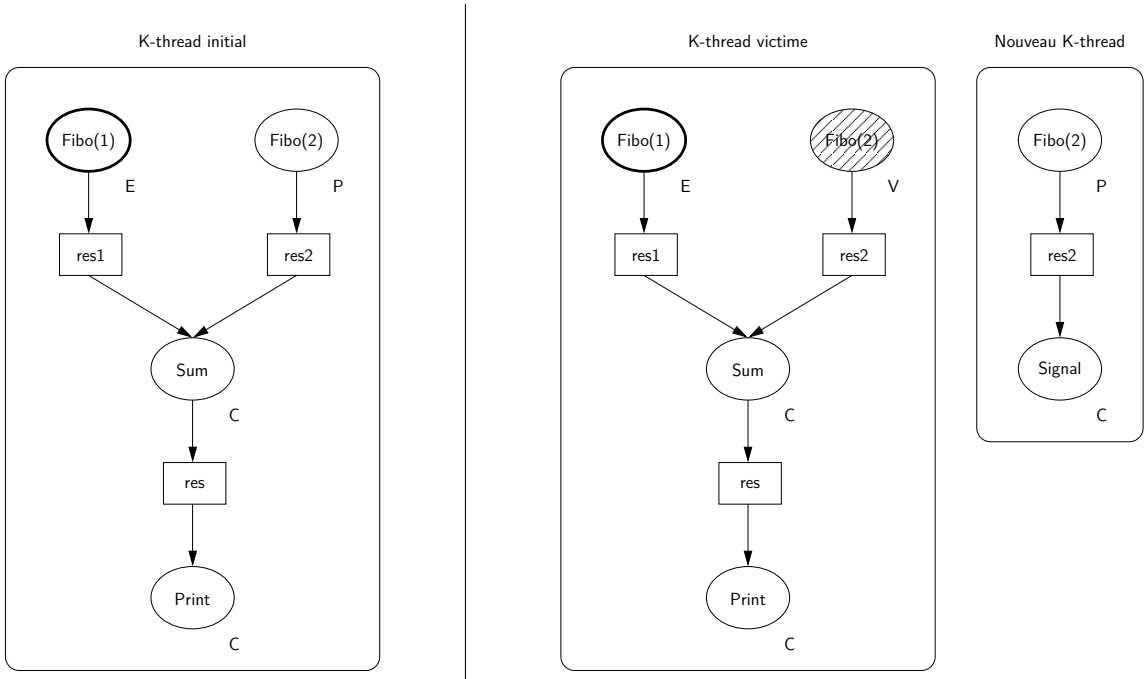
Lors de l'opération de vol de travail, le K-thread initial est découpé en deux K-threads indépendants qui se synchronisent sur la fin d'exécution de la tâche volée. Ils peuvent donc être exécutés en parallèle sur deux K-processeurs différents. Les dépendances entre les deux K-threads sont exprimées sous forme de communications. Il est important de constater que toutes les informations nécessaires à la synchronisation sont inscrites dans le graphe, sous forme d'état sur les tâches, sous forme de dépendances ou sous forme de tâches de communication.

La figure 4.8 montre comment le découpage par vol de travail s'applique sur le graphe de flot de données. Sur les graphes présentés dans cette figure, la lettre en bas à droite d'une tâche désigne son état. La sous-figure (a), à gauche montre le graphe de flot de données avant l'opération de vol de tâche. Le graphe utilisé pour l'exemple est celui du calcul de Fibonacci dont le programme est donné à la figure 4.2b. La sous-figure (b), à droite, montre le graphe de flot de données résultant de l'opération de vol. La tâche `Fibo(2)` a été volée ; elle est marquée Volée dans le K-thread victime et elle est recopiée dans un nouveau K-thread. Une tâche `Signal` est également ajoutée pour permettre la synchronisation des deux K-threads.

Une opération de vol peut être locale ou distante. Un vol distant signifie que le K-processeur voleur et le K-thread victime sont situés sur des processus différents. Dans ce cas, l'opération de découpage est suivie de la migration du nouveau K-thread vers le K-processeur voleur.

À l'opposé, un vol local signifie que le K-processeur voleur et le K-thread victime sont tous les deux sur le même processus. Dans ce cas, l'opération de vol et le retour de vol peuvent être optimisés pour limiter les copies et éviter les communications.

Une opération de vol de travail peut présenter un cout non négligeable, particulièrement lors d'un vol distant. C'est pourquoi il est nécessaire d'amortir le cout d'une opération en volant une quantité suffisamment importante de travail. C'est pourquoi



(a) Avant l'opération de vol

(b) Après l'opération de vol

FIG. 4.8: Découpage d'un graphe de flot de données par vol de travail

les applications récursives sont bien adaptées à ce type d'ordonnancement puisque le vol d'une tâche peut potiellement générer beaucoup de travail. C'est le cas de l'exemple de Fibonacci proposé à la figure 4.8. L'exécution de la tâche `Fibo(2)` va développer de nouvelles tâches comme sur l'exemple de la figure 4.5c.

Dans le cadre des applications séries-parallèles, l'ordonnancement par vol de travail offre une bonne répartition de charge. Des modèles de cout et des garanties de performances ont été donnés dans [39, 77].

4.3.2.3 Partitionnement statique

Le partitionnement statique est un ordonnancement qui vise à partitionner et à distribuer une sous-partie d'un graphe de flot de données pour l'exécuter en parallèle. Ceci correspond à prendre un K-thread ou une partie d'un K-thread et à le découper en p K-threads pour l'exécuter sur p K-processeurs. Le partitionnement statique est présenté en détail dans [117]. L'opération de partitionnement statique se compose de trois étapes : partitionnement, génération et distribution.

Partitionnement Le partitionnement est l'étape qui est chargée d'affecter un site d'exécution à chaque tâche du graphe de flot de données considéré. Pour cela, KAAPI repose sur des bibliothèques externes d'ordonnancement ou de partitionnement. Actuellement, les algorithmes suivants ont été intégrés à KAAPI :

- DSC [161] (*Dominant Sequence Clustering*) cherche à regrouper les tâches d'un graphe de précédence de manière à minimiser le temps d'exécution. DSC propose un ordonnancement pour un nombre illimité de processeurs. Un algorithme

d'équilibrage de charge [160] est utilisé pour replier les k partitions données par DSC sur les p processeurs disponibles.
- ETF [91] (*Earliest Task First*) calcule l'ordonnancement de tâches à partir d'un graphe de précédence sur un nombre p donné de processeurs. Pour cela, il cherche à calculer la date de démarrage au plus tôt de chaque tâche.
- METIS [95] agglomère les tâches d'un graphe de dépendances pondéré dans le but de générer des partitions équilibrées. METIS peut prendre en compte le cout des tâches et le cout des communications. Les tâches qui communiquent beaucoup sont regroupées dans une même partition.
- SCOTCH [115] fonctionne de manière similaire à METIS. Cependant il est capable de partitionner pour une machine cible hétérogène (en puissance et en bande passante). La machine cible est alors décrite sous forme d'un graphe non orienté annoté.

Une comparaison de ces algorithmes est effectuée dans [117].

L'étape de partitionnement nécessite de convertir le graphe de flot de données initial dans le format d'entrée de la bibliothèque utilisée. Le résultat rendu par la bibliothèque de partitionnement est utilisé pour ajouter un attribut de site logique d'exécution sur le graphe de flot de données initial. Ce site logique d'exécution correspond au numéro de partition.

Génération La génération est l'étape qui va effectivement découper le graphe de flot de données considéré en p sous-graphes de flot de données. Pour cela, la génération tient compte des numéros de partition donnés par l'étape de partitionnement.

Dans le graphe initial, lorsqu'une version est produite sur une partition i et utilisée sur une partition j, l'algorithme de génération doit ajouter une communication. Ainsi la génération remplace certains arcs entre les tâches et les versions par une communication en utilisant les tâches spéciales suivantes :
- La tâche `Broadcast` est la tâche d'émission qui prend un accès en lecture à une donnée partagée. Cette tâche est placée dans la même partition que la tâche qui produit la version de la donnée. L'exécution de cette tâche provoque la diffusion de la version de la donnée à toutes les autres partitions qui possèdent une tâche qui accède en lecture à cette version de la donnée. Ces partitions destinataires sont indiquées à l'aide d'un attribut sur la tâche `Broadcast`. Pour permettre un meilleur recouvrement calcul/communications, l'émission de la donnée par cette tâche est asynchrone.
- La tâche `Wait` est la tâche d'attente de fin d'émission qui est insérée après la tâche `Broadcast` et qui prend un accès en lecture à la même version que celle lue par la tâche `Broadcast`. Cette tâche est insérée dans l'état Attente et elle est débloquée par la fin de l'émission de la donnée. Son exécution ne fait rien mais elle permet de garantir que la version de cette donnée ne sera pas écrasée par une écriture avant que l'émission de la donnée ne soit terminée. Tant qu'une tâche en lecture existe sur cette donnée, elle ne pourra pas être modifiée[7]. Ceci est nécessaire car l'émission est réalisée de manière asynchrone.

[7]Un accès en écriture sur cette donnée peut quand même être exécuté, mais il utilisera un espace mémoire différent pour écrire sa valeur.

- La tâche **Receive** est la tâche de réception qui prend un accès en écriture à une donnée partagée. Elle est placée dans la même partition qu'une tâche qui lit une version produite dans une autre partition. Cette tâche est créée dans l'état Attente et elle est passée dans l'état Créée par la réception de la donnée. Son exécution ne fait rien, mais elle permet de bloquer l'exécution des tâches qui utilisent cette version de la donnée en attendant sa réception.

Pour résumer, la tâche **Broadcast** déclenche l'émission de la donnée et la tâche **Wait** correspond à la fin de l'émission de la donnée. Entre ces deux tâches, d'autres tâches peuvent être exécutées pour recouvrir la communication par du calcul. La tâche **Receive** correspond à la fin de la réception de la donnée. Pour identifier chaque communication, les tâches **Broadcast**, **Wait** et **Receive** sont associées à l'aide d'un identifiant.

De plus, l'étape de génération ajoute au début et en fin de chaque K-threads une phase de redistribution des données d'entrée et de sortie du graphe initial. Au début de chaque K-thread, des tâches de réception **Receive** sont ajoutées pour recevoir les données qui se trouvaient potentiellement sur un autre processus. De même, à la fin de chaque K-thread, des tâches d'émission **Broadcast** et **Wait** sont ajoutées pour émettre les données de sortie qui peuvent être utilisées sur d'autres processus. Ces tâches sont ajoutées de façon systématique de manière à obtenir des K-thread indépendants d'un placement particulier ou d'un algorithme de redistribution particulier des données.

Par exemple, si tous les K-threads sont finalement associés au même K-processeur, ces tâches de redistribution de données ne sont pas nécessaires et donc leur exécution consistera juste à leur changement d'état. Par contre, il est toujours possible de migrer ces K-threads sur des processus distants et dans ce cas, il n'y a pas de modification à apporter aux K-threads.

À la fin de l'étape de génération, on obtient p K-threads qui représentent le même calcul que le graphe de flot de données initial. Pour permettre leur distribution, les dépendances entre les tâches ont été remplacées par des communications grâce aux tâches **Broadcast**, **Wait** et **Receive**.

Distribution La distribution est l'étape qui va distribuer les p K-threads générés sur les K-processeurs. Pour cela, un algorithme de placement va d'abord associer à chaque site logique (numéro de partition) un site physique d'exécution (un K-processeur). Ensuite, les K-threads sont distribués en fonction de ce placement.

En pratique, on peut choisir de distribuer les p K-threads sur un nombre moindre de K-processeurs. Dans ce cas, plusieurs algorithmes de placement sont proposés. Certains font un placement cyclique (les K-threads k, $2k$, ... sont placés sur le K-processeur k), d'autres proposent un placement par blocs (les K-threads 1 à $k-1$ sont placés sur le K-processeur 1, les K-threads k à $2k-1$ sur le K-processeur 2, etc.). L'algorithme de placement peut aussi tirer parti de la représentation abstraite de la plate-forme d'exécution ou des graphes de flot de données à exécuter (pour tenir compte du volume de données à échanger entre deux K-threads).

Proposition 4 *L'opération du partitionnement statique est une* transformation valide.

Preuve Nous donnons l'idée de la démonstration. Le principe de cette transformation est de regrouper les tâches en fonction d'une décision de placement sur une machine particulière. L'ordre entre tâche du graphe avant le partitionnement est conservé après transformation : les tâches de communication, soumises aux contraintes de flot de données locales, permettent d'émettre les valeurs de données qui auraient été produites sans partitionnement.

Les tâches de réception marquent le fait qu'une donnée n'est pas encore produite localement : celle-ci sera activée sur réception du message contenant la donnée. Les tâches qui suivent une tâche de réception verront la même valeur que celle qui aurait été produite sans partitionnement. □

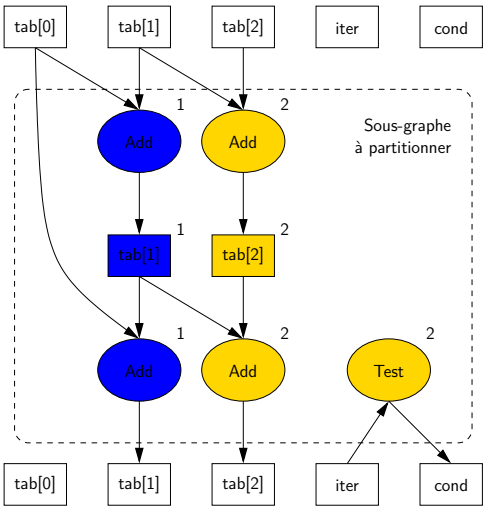


FIG. 4.9: Graphe de flot de données correspondant au développement de deux tâches `Kernel` et d'une tâche `Test` du programme de la figure 4.4b

La figure 4.9 montre le graphe de flot de données correspondant au développement de deux tâches `Kernel` et d'une tâche `Test` du programme d'exemple Pascal de la figure 4.4b. Une sous-partie du graphe a été partitionnée. Deux partitions ont été définies ; la partition 1 de couleur bleue et la partition 2 de couleur jaune. L'attribut du numéro de partition est montré en haut à droite d'une tâche ou d'une donnée.

La figure 4.10 montre les deux K-threads qui ont été générés après le partitionnement de la figure 4.9. Le K-thread généré 1 reprend les tâches de la partition bleue tandis que le K-thread généré 2 reprend les tâches de la partition jaune. Ces deux K-threads sont augmentés des tâches de communication nécessaires pour une exécution distribuée. Pour simplifier la figure, les tâches `Wait` n'apparaissent pas.

Les redistributions d'entrée et de sortie, encadrées en gris, gèrent les données d'entrée et de sortie du sous-graphe partitionné. Dans la figure 4.9, une version de la donnée `tab[1]` est produite sur la partition bleue et est utilisée sur les deux partitions. Cela implique l'ajout d'une communication entre les deux partitions qui est gérée par les tâches `Broadcast` et `Receive` que l'on peut voir sur la figure 4.10.

Une opération de partitionnement statique peut être couteuse puisqu'elle peut manipuler un nombre de tâches et de données important. De même la redistribution

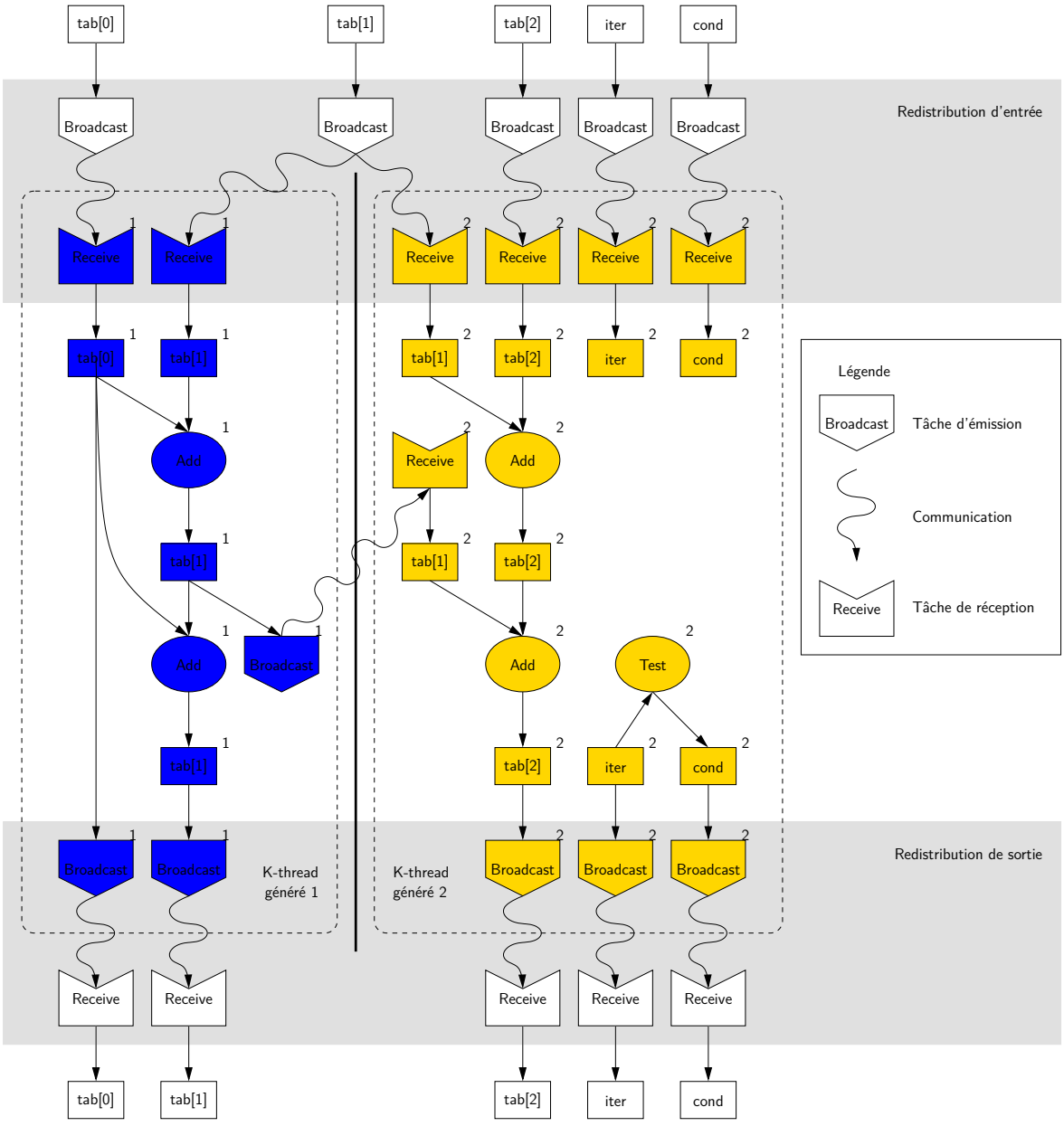


FIG. 4.10: Graphes de flot de données obtenus par partitionnement statique du graphe de la figure 4.9

des données d'entrée et de sortie aura un impact notable si le volume des données est conséquent. Les couts des opérations de partitionnement et de redistribution doivent donc être amortis.

Les algorithmes itératifs utilisant l'instruction ATHAPASCAN `While` sont bien adaptés au partitionnement statique car à chaque itération, le partitionnement sera le même[8]. Dans ce cas, le moteur d'exécution KAAPI propose un système de mise en cache des K-threads générés pour pouvoir exécuter les applications itératives sans recalculer le partitionnement statique à chaque itération.

De même, les redistributions d'entrée et de sortie sont programmées (en modifiant les attributs des tâches d'émission) pour tenir compte du caractère itératif de l'application. En particulier pour les applications itératives présentant une forte localité des données[9], les redistributions de données effectives sont faibles, voire nulles. Sur l'exemple de la figure 4.10, l'enchainement de plusieurs itérations du calcul ne provoquera, à chaque itération, que la redistribution de la donnée `tab[1]`, produite sur le K-thread 1 et utilisée sur le K-thread 2.

Le détail de ces mécanismes de contrôle des itérations et de redistribution est présenté dans [117].

4.4 Conclusion

Ce chapitre a présenté l'interface de programmation ATHAPASCAN et son extension. Le fonctionnement du moteur d'exécution KAAPI qui permet d'exécuter des applications ATHAPASCAN a été décrit. KAAPI fournit un fonctionnement séquentiel où un K-thread, représentant le calcul de l'application, est exécuté par un K-processeur, représentant une ressource de calcul. Un K-thread correspond à un graphe de flot de données qui sert de représentation abstraite du calcul de l'application. Cette représentation abstraite est observable, manipulable et migrable.

L'exécution parallèle est réalisée en découpant les graphes de flot de données grâce au parallélisme exprimé à l'aide du langage ATHAPASCAN. Plusieurs K-threads peuvent alors s'exécuter en parallèle et en distribué, sur plusieurs K-processeurs de manière transparente. Les synchronisations sont automatiquement gérées par des communications.

Deux opérations de découpage sont proposées : le vol de travail, plutôt adapté aux applications récursives, et le partitionnement statique, plutôt adapté aux applications itératives. Ces opérations de découpage ne modifient pas la sémantique du programme ATHAPASCAN.

Il faut noter que le langage ATHAPASCAN permet de décrire une application avec une sémantique de lecture séquentielle du programme. De plus, aucune communication n'est spécifiée explicitement dans le modèle de programmation ATHAPASCAN. Grâce au graphe de flot de données, les communications sont déduites automatiquement des dépendances lors des opérations d'extraction de parallélisme.

[8] Sous réserve que le nombre de K-processeurs disponibles ne change pas.

[9] C'est par exemple le cas des applications basées sur la décomposition de domaine.

La représentation abstraite sous forme d'un graphe de flot de données est un point clé dans le reste de ces travaux, autant pour la partie reconfiguration dynamique que pour la partie tolérance aux fautes.

Mécanismes de reconfiguration 5

Sommaire

5.1 Introduction

Ce chapitre propose des mécanismes de reconfiguration pour les applications parallèles distribuées. Ces mécanismes sont instanciés dans le moteur d'exécution Kaapi. Pour cela, nous plaçons le moteur d'exécution Kaapi tel qu'il a été présenté au chapitre précédent dans un contexte où nous distinguons deux modes de fonctionnement. Un mode de fonctionnement *normal* où chaque K-processeur exécute un K-thread séquentiellement selon l'ordre de référence et un mode de reconfiguration où une opération de reconfiguration est appliquée sur l'application.

Ainsi, nous choisissons de considérer les opérations d'extraction de parallélisme (le vol de travail et le partitionnement statique) présentées au chapitre précédent comme des opérations de reconfiguration de l'application. D'autres opérations de reconfiguration peuvent également être envisagées comme la migration de K-threads ou les opérations de

tolérance aux fautes. La section 7.3 du chapitre 7 présentera un protocole de sauvegarde coordonnée reposant sur ces mécanismes de reconfiguration.

5.1.1 Types de reconfigurations

Pour différencier les reconfigurations possibles, nous utilisons plusieurs critères. Tout d'abord, nous distinguons les reconfigurations du moteur d'exécution et les reconfigurations de l'application.

Une **reconfiguration du moteur d'exécution** change la manière dont le graphe de flot de données est exécuté, cependant elle ne doit pas changer la sémantique de l'application. Plus précisément, une reconfiguration qui modifie la sémantique du graphe de flot de données n'est pas valide puisque dans ce cas, le moteur d'exécution ne réalise plus correctement sa fonction. Par les reconfigurations valides, nous trouvons par exemple le changement du réseau utilisé pour communiquer entre deux processus donnés (par exemple Myrinet au lieu de TCP). Ce type de reconfiguration est spécifique au moteur d'exécution Kaapi. Elles ne font pas l'objet d'étude de nos travaux, bien que pouvant aussi bénéficier des mécanismes de reconfiguration que nous proposons.

Les reconfigurations de l'application sont celles visées par ces travaux. Une **reconfiguration de l'application** est une reconfiguration qui va agir sur la représentation abstraite de l'application, *i.e.* son graphe de flot de données. Une telle reconfiguration peut être indépendante ou dépendante de l'application :

- Une **reconfiguration indépendante de l'application** est une reconfiguration qui garantit que les opérations faites sur le graphe sont des modifications qui respectent la sémantique Athapascan. Les opérations effectuées reposent uniquement sur la définition du graphe de flot de données et la sémantique de lecture/écriture associée ; elles ne font aucune hypothèse sur les calculs exécutés par les tâches. Voici des exemples de telles reconfigurations :
  - les opérations de découpage de graphe de flot de données présentées dans le chapitre précédent (le vol de travail, section 4.3.2.2 ; et le partitionnement statique, section 4.3.2.3) ;
  - une opération de fusion qui regroupe les tâches des deux K-threads en un seul ;
  - une opération de changement de l'ordre de référence[1] des tâches d'un K-thread.
- Une **reconfiguration dépendante de l'application** est définie en se basant sur une connaissance ou une propriété de l'application. Ces reconfigurations font des modifications qui changent la sémantique du graphe Athapascan. Elles sont dédiées à une application particulière et sont généralement définies par l'utilisateur de l'application. Parmi ces reconfigurations, nous donnons les quelques exemples suivants :
  - le remplacement d'une tâche par une autre qui calcule un résultat approché mais avec une méthode plus performante ;
  - le découpage d'une tâche en plusieurs tâches pour exposer plus de parallélisme, ou inversement la fusion de plusieurs tâches en une seule ;

[1] L'ordre de référence n'apparait pas dans la représentation sous forme de graphe de flot de données, cependant il correspond à l'ordre d'empilement des tâches dans le K-thread. Il est utilisé comme ordre d'exécution par défaut par le K-processeur et peut influencer les performances de l'application.

– l'ajout de tâches destinées à afficher la valeur de données.
Ces opérations dépendent explicitement des opérations effectuées par les tâches impliquées dans la reconfiguration.

Orthogonalement, nous distinguons également les reconfigurations locales et les reconfigurations distribuées. Pour cela, nous regardons la notion de distribution du moteur d'exécution KAAPI pour lequel le grain de distribution est le K-thread. En effet dans KAAPI, un K-thread est un élément toujours exécuté séquentiellement, tandis que deux K-threads peuvent s'exécuter sur deux processeurs différents, donc potentiellement de manière distribuée.

Nous définissons donc une **reconfiguration locale** comme une reconfiguration qui n'opère que sur un seul K-thread ou sur seulement une partie d'un K-thread. Inversement, une **reconfiguration distribuée** est une reconfiguration qui opère sur des éléments d'au moins deux K-threads différents.

5.1.2 Problématiques

Les mécanismes de reconfiguration qui font l'objet de ce chapitre ont été conçus pour le moteur d'exécution KAAPI. Cependant ils sont présentés de manière à être indépendants, autant que possible, du moteur d'exécution KAAPI et de son implémentation. Pour cela, nous nous plaçons dans le modèle suivant, qui est calqué sur KAAPI.

- Notre application est constituée d'un ensemble de processus distribués.
- Ces processus communiquent par l'intermédiaire de canaux de communication supposé FIFO (*First In, First Out*).
- L'état de l'application est représenté par une structure de données abstraite qui est distribuée sur l'ensemble des processus.
- Cette structure de données est constituée d'un ensemble d'objet indépendants qui représentent les calculs à effectuer.
- L'exécution est réalisée en interprétant les objets de la représentation abstraite de l'application.
- Une reconfiguration est réalisée en modifiant l'état de l'application, c'est-à-dire en modifiant les objets qui constituent la représentation abstraite.

Pour concevoir ce mécanisme de reconfiguration pour le moteur d'exécution KAAPI, nous nous sommes intéressés aux problématiques suivantes : les accès concurrents et la gestion de la cohérence entre les objets de la représentation abstraite.

Gestion des accès concurrents. Les reconfigurations que nous considérons sont dynamiques, c'est-à-dire qu'elles peuvent être déclenchées à tout moment de l'exécution. De plus, le déclenchement de ces reconfigurations est asynchrone ; il est exécuté dans son propre processus léger, *i.e.* son propre *thread* noyau. Ce genre de comportement correspond au cas où la reconfiguration peut être déclenchée par une communication, un évènement périodique ou une temporisation[2].

Ainsi, les processus légers qui exécutent l'application et le processus léger qui exécute la reconfiguration peuvent s'exécuter en concurrence. Si les deux processus

[2]Dans KAAPI, les communications et le démon d'évènements utilisent des *threads* noyau dédiés.

légers accèdent et modifient un même objet en même temps, cela peut provoquer des incohérences.

Gestion de la cohérence. Cette problématique a été présentée dans l'état de l'art à la section 3.3.3. Elle est généralement décomposée en trois aspects : l'intégrité structurelle, la cohérence mutuelle et les invariants d'état de l'application. Pour la suite, nous nous focalisons principalement sur la cohérence mutuelle.

La problématique de la cohérence mutuelle intervient lorsque la reconfiguration cible deux objets qui peuvent interagir. Lorsque deux objets entrent en interaction, l'état des objets peut être lié. Dès lors, pour appliquer correctement une reconfiguration, il peut être nécessaire de garantir la cohérence mutuelle entre l'état observé de ces objets. Il faut ajouter deux remarques.

- Tout d'abord, la notion de cohérence mutuelle n'a de sens que lorsque la reconfiguration opère sur au moins deux objets distribués[3]. Cela correspond aux reconfigurations distribuées.
- De plus, la cohérence mutuelle n'est pas une propriété indispensable pour toutes les reconfigurations. La cohérence mutuelle peut-être nécessaire à certaines reconfigurations mais pas pour d'autres reconfigurations.

L'objectif de ce chapitre est donc d'offrir une solution aux problématiques exposées précédemment. Ce chapitre est organisé de la manière suivante. La prochaine section présente une modélisation du processus de reconfiguration. La section 5.3 présente le mécanisme gestion des accès concurrents et la section 5.4 le mécanisme de gestion de la cohérence.

5.2 Modélisation du processus de reconfiguration

5.2.1 Définition d'une reconfiguration

Nous définissons une reconfiguration de l'application comme une opération qui modifie l'état de l'application. Dans notre cas, l'état de l'application est représenté par une structure de données abstraite (un graphe de flot de données dans le cas de Kaapi). Une reconfiguration peut être vue comme une fonction qui est appliquée sur plusieurs objets pour les modifier.

Une opération de reconfiguration est définie par sa **fonction de reconfiguration**. C'est la fonction qui sera exécutée par chacun des processus de la cible de la reconfiguration. Cette fonction reconfigure l'application en modifiant l'état de chaque processus (c'est-à-dire qu'elle modifie le graphe de flot de données pour les processus Kaapi). Elle est écrite en langage de programmation classique et elle peut prendre des paramètres en entrée. Elle retourne également une valeur de sortie.

[3]Ici distribué signifie que les deux objets sont affectés par au moins deux flots d'exécution, mais pas forcément distant. En effet, sur une même machine, une modification effectuée par un processeur n'est pas forcément visible instantanément par un autre processeur. On peut observer alors un délai similaire à une communication qui est dû à un réordonnancement des instructions d'écriture et de lecture par le compilateur ou le processeur.

L'**instanciation d'une reconfiguration** est réalisée en donnant, en plus de la fonction de reconfiguration définie ci-dessus, les éléments suivants.

- La **cible de la reconfiguration**, qui correspond à l'ensemble de processus sur lesquels doit être exécutée la fonction de reconfiguration. Dans le cas du moteur d'exécution KAAPI, les processus sont désignés grâce à un identifiant unique.
- Les **valeurs des paramètres d'entrée** de la fonction de reconfiguration.

5.2.2 Déroulement d'une reconfiguration

La fonction de reconfiguration est exécutée comme un programme SPMD[4] sur l'ensemble des processus cibles. On distingue trois étapes dans le déroulement d'une reconfiguration : le prologue, la réalisation et l'épilogue.

Prologue. Le prologue (ou déclenchement) est l'étape qui réalise l'instanciation de la reconfiguration en définissant les différents éléments qui la compose (la fonction, la cible et les paramètres). Le prologue est un programme séquentiel qui peut s'éxécuter sur n'importe quel processus et dans n'importe quel flot d'exécution. Le processus qui exécute le prologue (c'est-à-dire celui qui déclenche la reconfiguration) est appelé le **processus maitre** de la reconfiguration. Le prologue est terminé par l'**invocation de la reconfiguration**, qui provoque la réalisation de la reconfiguration. L'appel de l'invocation bloque le flot qui exécute le prologue jusqu'à la fin de la réalisation.

Réalisation. La réalisation est l'étape durant laquelle la fonction de reconfiguration est exécutée sur chacun des processus qui composent la cible de la reconfiguration. Le processus léger qui exécute cette fonction est appelé **flot de reconfiguration**. Selon l'implémentation (*cf* section 5.3), le flot de reconfiguration peut être concurrent avec l'exécution de l'application. Par défaut, il n'y a pas de contrainte de synchronisation entre les flots de reconfiguration des différents processus cibles. Cependant, l'ajout d'une contrainte de cohérence mutuelle (*cf* section 5.4.3) ou l'utilisation explicite de communication entre les processus peut être utilisée pour synchroniser les processus. À la fin de l'exécution de la fonction de reconfiguration, chaque processus retourne le résultat de la fonction au processus maitre de la reconfiguration.

Nous distinguons deux fonctions essentielles qui composent la phase de réalisation d'une reconfiguration : l'**exploration** et la **mutation**. L'exploration désigne les accès en lecture du flot de reconfiguration, tandis que la mutation désigne les modifications des objets par le flot de reconfiguration. L'exploration permet au flot de reconfiguration d'explorer et d'analyser l'état de l'application. Le but de l'exploration est de trouver et de marquer les objets à modifier, c'est-à-dire de désigner les objets qui seront modifiés par la mutation. La mutation permet alors au flot de reconfiguration de réellement modifier les objets qui ont été marqués pour réaliser la reconfiguration.

Épilogue. L'épilogue est l'étape qui correspond à la fin de la réalisation et à la reprise du flot d'exécution qui a effectué l'invocation. L'épilogue commence lorsque la fonction de reconfiguration a fini d'être exécutée sur tous les processus cibles et que tous les

[4] *Single Program, Multiple Data*

résultats ont été retournés. L'épilogue permet d'analyser les résultats de la réalisation de la reconfiguration.

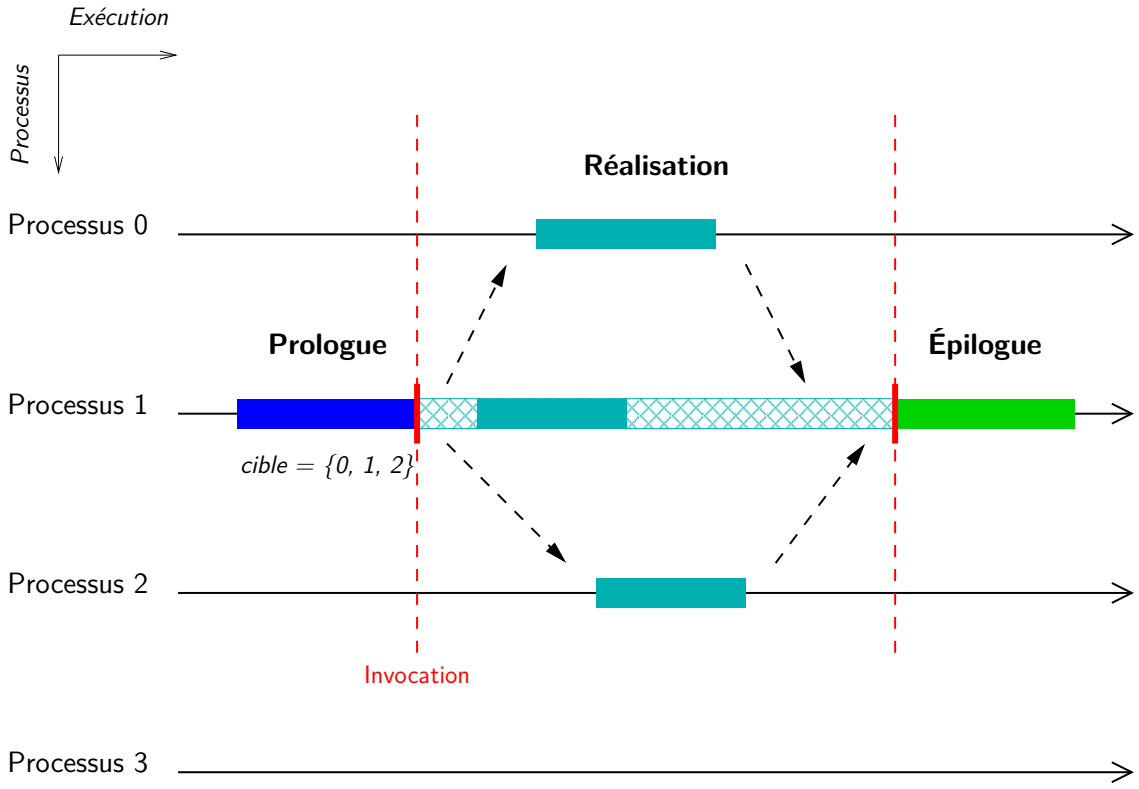


FIG. 5.1: Déroulement d'une reconfiguration distribuée

La figure 5.1 montre le déroulement d'une reconfiguration sur un exemple. L'application est composée de quatre processus et l'axe horizontal indique l'avancée de l'exécution. Le processus 1 est le processus maitre qui instancie la reconfiguration en définissant l'ensemble des processus 0, 1 et 2 comme cible de la reconfiguration.

Si deux reconfigurations sont réalisées simultanément, plusieurs processus peuvent être impliqués dans les deux reconfigurations. Dans ce cas, il faut garantir que les reconfigurations seront appliquées dans le même ordre sur chacun des processus. Cela peut être réalisé grâce à un algorithme d'exclusion mutuelle répartie [128].

5.3 Gestion des accès concurrents

La gestion des accès concurrents vise à protéger les objets cibles de la reconfiguration de toutes modifications extérieures durant la durée de la reconfiguration. Dans la suite de cette section, nous présentons d'abord la notion d'objet reconfigurable et une modélisation des contraintes liées aux flots d'exécution et de reconfiguration. Puis nous proposons deux méthodes d'exécution qui satisfont ces contraintes : l'exécution concurrente et l'exécution coopérative.

5.3.1 Notion d'objet reconfigurable

L'état de l'application est constitué par une structure de données appelée **représentation abstraite** de l'application. Elle est constituée d'objets indépendants qui représentent les calculs à effectuer. Une reconfiguration est une opération qui modifie un ou plusieurs objets qui constitue l'état de l'application.

Nous définissons la notion d'**objet reconfigurable** ou **R-objet** comme un objet de la représentation abstraite de l'application qui peut être modifié de manière cohérente indépendamment des autres. Pour cela, l'état de cet objet doit être clos (*i.e.* indépendant de tout autre structure de données) et accessible. Un R-objet est une encapsulation abstraite des structures de données physiques (c'est-à-dire telle qu'elles sont en mémoire). Un R-objet peut également être un conteneur qui contient d'autres R-objets. Ceci permet de refléter des hiérarchies d'objets en mémoire.

Par la suite, nous associons à chaque R-objet un état qui permet de connaitre comment les flots (d'exécution et de reconfiguration) accèdent à l'objet. Un R-objet est donc composé de deux parties, une partie qui contient l'état de l'objet de la représentation abstraite, et une partie qui contient l'état du R-objet vis-à-vis de la reconfiguration.

Structures de données et objets reconfigurables dans Kaapi. Pour Kaapi, nous définissons les objets reconfigurables en suivant l'organisation hiérarchique des données en mémoire. L'état d'une application dans le moteur d'exécution Kaapi est représenté comme un ensemble de K-threads répartis sur l'ensemble de processus Kaapi. Sur chaque processus, plusieurs K-processeurs ont pour rôle d'exécuter le calcul en interprétant le graphe de flot de données contenu dans les K-threads.

Dans Kaapi, la représentation interne de l'état de l'application sous forme d'un graphe de flot de données, repose sur les objets reconfigurables suivants [78].

- Le R-objet **K-thread** correspond à une partie du graphe de flot de données de l'application, selon la définition de la section 4.3. Il se comporte comme une pile de K-frames (ordre LIFO[5]).
- Le R-objet **K-frame** est l'équivalent pour les tâches Athapascan de la notion de *frame* (structure de données) d'une pile d'appel de fonctions lors de l'exécution d'un programme. Ainsi, un niveau de récursion dans le graphe de flot de données correspond à l'empilement d'une nouvelle K-frame dans le K-thread. Une K-frame se comporte comme une file de K-tasks (ordre FIFO[6]).
- Le R-objet **K-task** représente une tâche de calcul Athapascan. La structure de données K-task contient les informations sur les modes d'accès aux données partagées qui permettent de déduire les contraintes de dépendances du graphe de flot de données.

Cette structure de données est illustrée sur un exemple à la figure 5.2. Dans cette figure, les objets reconfigurables sont les K-threads, les K-frames et les K-tasks.

Le fonctionnement interne du moteur d'exécution est fortement couplé à la représentation en mémoire du graphe de flot de données et à la manière dont il est interprété. Ces structures de données ont été choisies pour optimiser l'exécution séquentielle (*Work*

[5] *Last In, First Out*
[6] *First In, First Out*

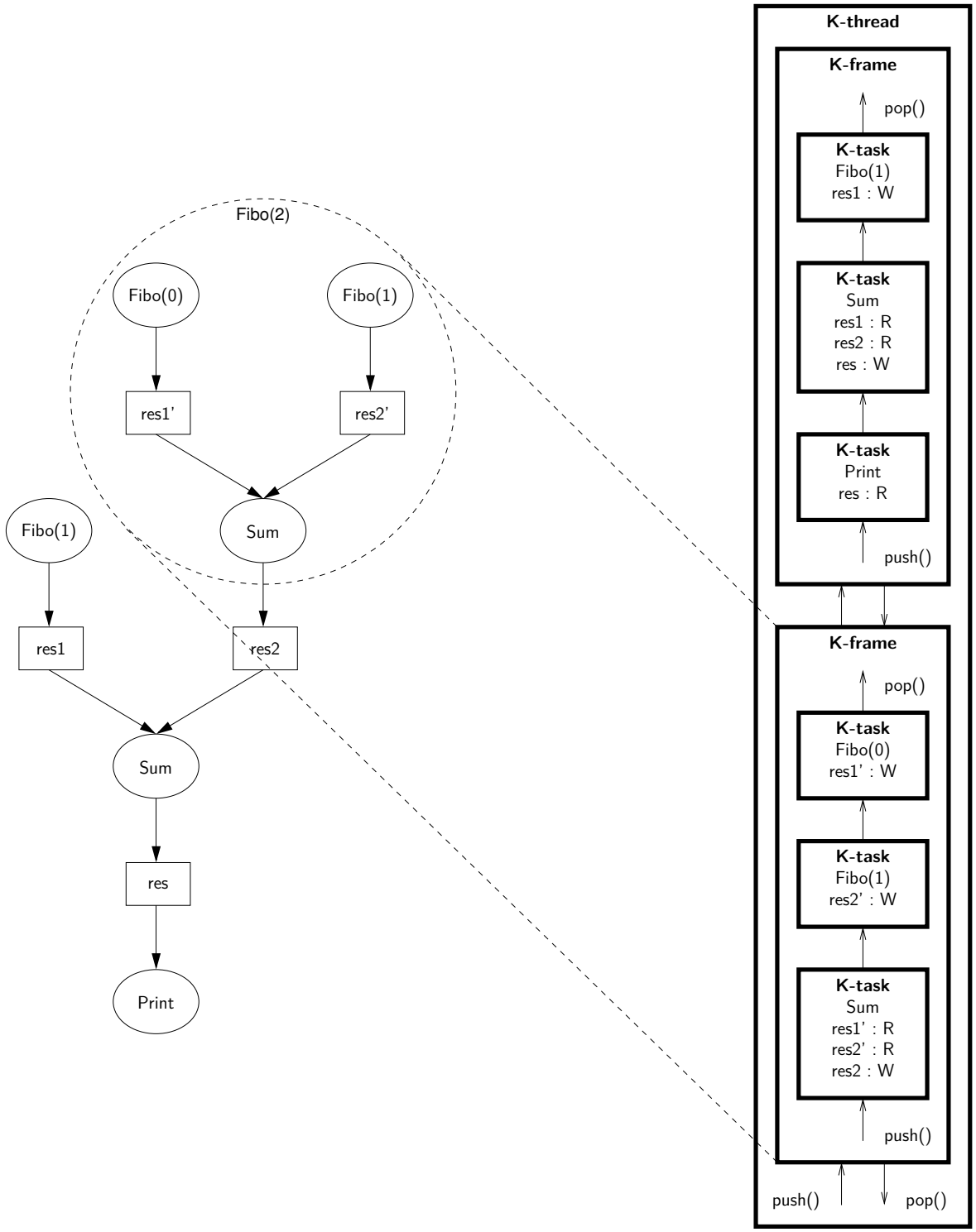


FIG. 5.2: Représentation interne du graphe de flot de données : les objets reconfigurables sont les K-threads, les K-frames et les K-tasks.

First Principle). Ainsi, l'ordre naturel de lecture des K-tasks (par les instructions `pop()`) correspond à l'ordre d'exécution suivi par le K-processeur (*i.e.* l'ordre de référence) qui est un ordre valide d'exécution respectant la sémantique ATHAPASCAN.

5.3.2 Flot d'exécution et flot de reconfiguration

Durant une exécution classique, sans reconfiguration, il y a au plus un flot d'exécution qui accède à un objet en mémoire puisque chaque K-thread est associé à un unique K-processeur. Cependant, durant une étape de reconfiguration, un nouveau processus léger, appelé flot de reconfiguration, est utilisé pour réaliser la reconfiguration. Ce flot de reconfiguration peut entrer en conflit avec les flots d'exécution de l'application. Pour cela, il est nécessaire de garantir l'exclusion mutuelle lorsque ces deux flots tentent de modifier un même objet.

Flot d'exécution. Durant l'exécution de l'application, chaque objet de la hiérarchie peut prendre deux états différents selon la présence d'un flot d'exécution. Le diagramme d'état d'un objet selon le flot d'exécution est donné à la figure 5.3.

L'action RUN est celle qui fait passer un objet Inactif dans l'état Exécution. Quand un objet est dans cet état, cela signifie qu'il est en cours d'exécution par le flot d'exécution. Un objet dans l'état Exécution peut être modifié par le flot d'exécution qui nécessite donc un accès exclusif à l'objet.

L'action STOP fait donc passer l'objet dans l'état Inactif. Dans cet état, le flot d'exécution n'accède pas à l'objet.

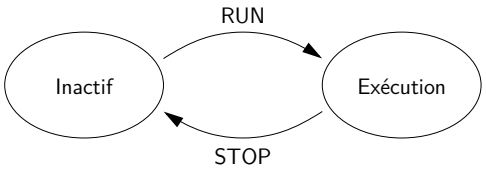


FIG. 5.3: Diagramme d'état d'un objet selon le flot d'exécution

Flot de reconfiguration. Lors de la réalisation d'une reconfiguration, un flot de reconfiguration est créé pour reconfigurer les objets. Un flot de reconfiguration peut réaliser deux types d'accès aux objets : des accès en lecture seule, c'est l'exploration et des accès de modification, c'est la mutation. Nous définissions donc trois états pour un objet selon le flot de reconfiguration. La figure 5.4 montre le diagramme d'état d'un objet selon le flot de reconfiguration.

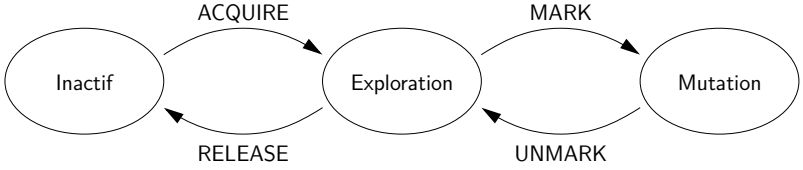


FIG. 5.4: Diagramme d'état d'un objet selon le flot de reconfiguration

L'état Inactif signifie que le flot de reconfiguration n'accède pas à l'objet. Les actions ACQUIRE et RELEASE permettent au flot de reconfiguration d'obtenir et de libérer une référence sur un objet. Ces actions font passer l'état de l'objet de Inactif à Exploration et inversement. L'état Exploration signifie que le flot de reconfiguration accède à l'objet en lecture. Le flot de reconfiguration ne peut pas modifier l'objet dans cet état ; mais comme l'objet est accédé en lecture, l'objet ne peut pas être détruit.

Les actions MARK et UNMARK permettent au flot de reconfiguration de marquer un objet qu'il va modifier. Ces actions font passer l'état de l'objet de l'état Exploration à l'état Mutation et inversement. Lorsque l'objet est dans l'état Mutation, le flot de reconfiguration peut modifier l'objet et donc il nécessite un accès exclusif à l'objet.

5.3.3 Exécution concurrente

La méthode d'exécution concurrente d'une reconfiguration consiste à laisser le flot de reconfiguration s'exécuter en concurrence du flot d'exécution. Cependant pour garantir la cohérence des objets de l'application, il faut assurer des accès exclusifs aux objets lorsqu'ils sont modifiés.

Lors d'une exécution concurrente d'une reconfiguration, un flot d'exécution et un flot de reconfiguration s'exécutent en même temps. Dans ce cas, l'état d'un objet est le produit cartésien de l'état de l'objet selon le flot d'exécution par l'état de l'objet selon le flot de reconfiguration. L'analyse de tous les états permet de détecter les cas de conflit, et donc d'éliminer les transitions qui ne respectent les accès exclusifs à l'objet.

Pour désigner l'état produit, nous utilisons la notation sous forme d'une paire (A, B), où la première composante A représente l'état de l'objet selon le flot d'exécution et la deuxième composante B représente l'état de l'objet selon le flot de reconfiguration.

- (Inactif, Inactif) désigne le cas où aucun flot n'accède à l'objet. Il n'y a pas de conflit.
- (Inactif, Exploration) désigne le cas où seul le flot de reconfiguration accède en lecture à l'objet. Il n'y a pas de conflit.
- (Inactif, Mutation) désigne le cas où seul le flot de reconfiguration accède à l'objet et qu'il le modifie. Il n'y a pas de conflit.
- (Exécution, Inactif) désigne le cas où seul le flot d'exécution accède à l'objet et le modifie. Il n'y a pas de conflit.
- (Exécution, Exploration) désigne le cas où le flot de reconfiguration accède à l'objet en lecture et que le flot d'exécution modifie l'objet. Il n'y a pas de conflit, cependant l'objet ne doit pas être détruit tant que le flot de reconfiguration y accède.
- (Exécution, Mutation) désigne le cas où le flot d'exécution et le flot de reconfiguration modifient tous les deux l'objet. Cet état est un état conflictuel car la modification par deux flots peut créer des incohérences dans l'état de l'objet. Il faut donc empêcher toute transition vers cet état.

La figure 5.5 montre le diagramme d'état résultant. Grâce à ce diagramme, nous pouvons connaitre les actions à entreprendre pour pouvoir reconfigurer un objet. Ainsi, lorsqu'un flot de reconfiguration veut modifier un objet, il doit d'abord y accéder en lecture, c'est la phase d'exploration. Pour modifier l'objet, le flot de reconfiguration doit le passer dans l'état (Inactif, Mutation). Deux cas sont alors possibles :

- L'objet est dans l'état (Inactif, Exploration), c'est-à-dire qu'il n'est pas en train

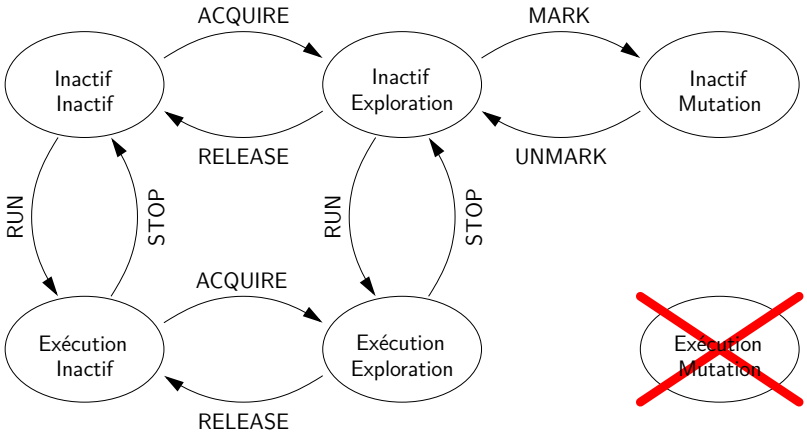


FIG. 5.5: Diagramme d'état d'un objet pour une exécution concurrente

d'être exécuté. Il peut directement passer dans (Inactif, Mutation) à l'aide de l'action MARK.

- L'objet est dans l'état (Exécution, Exploration), c'est-à-dire qu'il est en train d'être exécuté. Le flot d'exécution doit d'abord être arrêté, c'est l'action STOP. L'objet passe alors dans l'état (Inactif, Mutation) qui correspond au cas précédent.

La méthode d'exécution concurrente à l'avantage de permettre de reconfigurer des objets de l'application sans nécessairement arrêter tout le calcul.

Cependant, la méthode d'exécution concurrente nécessite l'utilisation de primitives de synchronisation pour garantir l'exclusion mutuelle entre le flot de reconfiguration et le flot d'exécution.

5.3.4 Exécution coopérative

Le principe de l'exécution coopérative est de faire coopérer le flot d'exécution et le flot de reconfiguration pour appliquer une reconfiguration. Le flot de reconfiguration sera exécuté par le même processus léger qui exécute le flot d'exécution de l'application. Le flot d'exécution doit préalablement être arrêté à l'aide de l'action STOP.

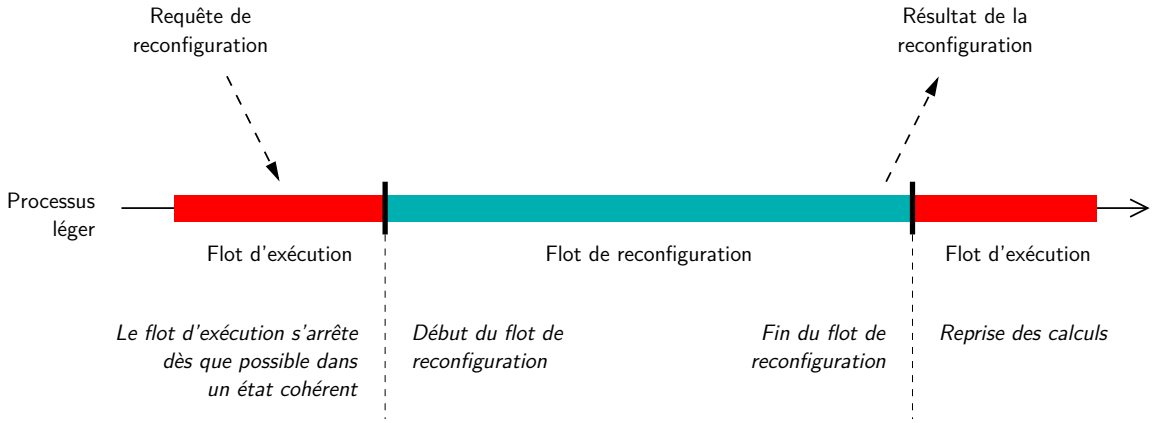


FIG. 5.6: Déroulement d'une reconfiguration par coopération

Le figure 5.6 montre le déroulement d'une telle coopération.

1. Tout d'abord, un signal de reconfiguration est émis vers le flot d'exécution qui exécute l'objet à explorer et à modifier.
2. Le flot d'exécution s'arrête alors dès que possible après avoir consolidé l'état de l'application, c'est-à-dire qu'il termine les modifications en cours pour laisser un état cohérent. Ceci correspond à la transition STOP du diagramme d'état.
3. Une fois que le flot d'exécution est arrêté, le processus léger exécute le flot de reconfiguration, sans interruption jusqu'à la fin.
4. Lorsque le flot de reconfiguration est terminé, le processus léger reprend l'exécution du flot d'exécution de l'application.

Avec une exécution coopérative, les flots d'exécution et de reconfiguration ne s'exécute jamais en même temps. Dans ce cas, le diagramme d'état d'un objet correspond à celui de la figure 5.7.

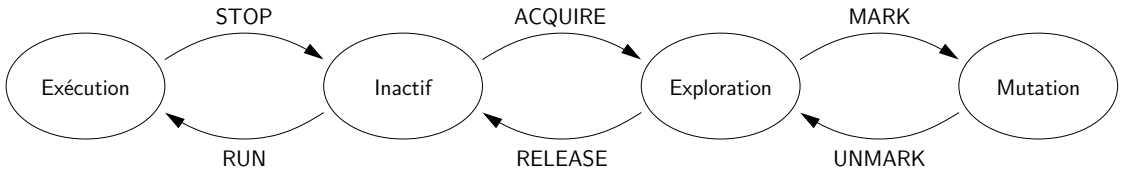


Fig. 5.7: Diagramme d'état d'un objet pour exécution coopérative

L'avantage de la méthode d'exécution coopérative est qu'elle ne nécessite pas d'utiliser des primitives de synchronisation pour garantir l'exclusion mutuelle sur les objets reconfigurables, ce qui permet de réaliser une implémentation plus efficace. Dans ce cas, la synchronisation est réalisée à un grain plus important par un mécanisme de signalisation pour l'émission et la réception de la requête de reconfiguration et du résultat.

5.3.5 Implémentation dans X-Kaapi.

X-Kaapi est une nouvelle implémentation du moteur d'exécution Kaapi qui vise à expérimenter, entre autres[7], le mécanisme d'exécution concurrente et coopérative. Pour le moment, ce prototype ne fonctionne qu'en mémoire partagée et ne traite que les reconfigurations « vol de travail ».

Exécution concurrente. L'implémentation actuelle du moteur d'exécution X-Kaapi offre un mécanisme de reconfiguration par exécution concurrente. Pour cela, les opérations de parcours et de manipulation de la structure de données qui représente l'état de l'application utilisent différentes primitives de synchronisation selon le type du R-objet considéré.

- Les R-objets de type K-thread et K-frame se comportent respectivement comme des piles et des files. L'exclusion mutuelle dans ces objets est garantie à la fois

[7]L'objectif de X-Kaapi est aussi de construire un noyau de calcul pour des systèmes embarqués qui soit petit et avec un surcoût à l'exécution minimum.

par des algorithmes sans verrou (l'insertion ne nécessite pas de prise de verrou) et par des verrous (en particulier pour la destruction).
- Les R-objets de type K-task sont sollicités beaucoup plus souvent que les autres R-objets (K-thread et K-frame) puisqu'ils sont les briques élémentaires de représentation du calcul. Les opérations de vérouillage et de dévérouillage étant couteuses, l'exclusion mutuelle sur les K-tasks est réalisée à l'aide de l'instruction `compare-and-swap` pour offrir de meilleures performances.

Lors d'une opération de vol de travail, lorsqu'un processeur est inactif, ce processeur déclenche une opération de reconfiguration qui s'exécute en concurrence avec les autres flots d'exécution du code applicatif ou de reconfiguration (dans le cas où d'autres processeurs seraient inactifs). Une tâche est volée ou exécutée de manière unique grâce à l'utilisation d'une opération de type `compare-and-swap`.

Exécution coopérative. L'objectif de cette exécution est de réduire au minimum l'utilisation des primitives de synchronisation pour améliorer les performances. Ainsi pour suivre le principe du travail d'abord (*Work first principle*), le code du flot d'exécution qui exécute les tâches créées (contenu dans les K-frames) ne comporte aucune synchronisation. Lors d'une opération de reconfiguration, ce flot est interrompu le temps du traitement (vol d'une tâche) de l'opération de reconfiguration. Une synchronisation existe entre les différents voleurs cherchant à voler un même flot d'exécution, mais aucune n'existe lors de l'exécution des tâches.

Ce type d'exécution possède un surcoût extrêmement faible dans le cas de l'exécution d'un programme parallèle à grain fin (le nombre d'instructions exécutées par une tâche est très faible ; le nombre de tâches est en $O(T_1)$) et ayant un chemin critique T_∞ très faible ($T_\infty \ll T_1$) :
- dans ce type d'ordonnnancement, le nombre de requêtes de vol est petit (de l'ordre de T_∞) [39, 80] ;
- en cas de requête de vol, le temps d'attente avant que l'opération de reconfiguration soit traitée est faible car la durée d'exécution des tâches est faible.

À l'inverse, l'implémentation par exécution concurrente dans X-Kaapi, une opération `compare-and-swap` est ajoutée pour chaque tâche exécutée.

5.4 Gestion de la cohérence

Comme nous l'avons vu dans la section 3.3.3 page 55 du chapitre 3, la gestion de la cohérence repose sur les trois points suivants : l'intégrité structurelle, la cohérence mutuelle et les invariants de l'état [110, 6]. Nous présentons comment nous garantissons ces trois aspects.

5.4.1 Intégrité structurelle

L'intégrité structurelle signifie que la structure de l'application doit respecter les contraintes des interfaces des objets et la manière dont ils sont connectées.

Notre modèle de reconfiguration est basé sur le moteur d'exécution Kaapi et le langage de programmation objet C++. De plus, les opérations de reconfiguration que nous proposons s'expriment sous forme de manipulation du graphe de flot de données et

de ses attributs. Nous reposons alors sur une interface de manipulation du graphe qui permet de vérifier les modifications effectuées par la reconfiguration. Ces vérifications sont alors effectuées de deux manières, statiquement à la compilation en imposant un typage sur les objets constituant le graphe et ses attributs, et dynamiquement à l'exécution à l'aide d'assertions insérées dans l'implémentation de l'interface de manipulation du graphe.

5.4.2 Invariants de l'état

Un invariant de l'état de l'application est un prédicat portant sur tout ou une partie du système. Il est exprimé en fonction des variables d'état de l'application. Il est, en général, intrinsèquement lié à l'application.

Dans le moteur d'exécution Kaapi, la représentation abstraite de l'état de l'application sous forme d'un graphe de flot de données permet d'exprimer facilement des invariants de l'état. De plus, comme cela a été présenté à la section 4.2.4 du chapitre 4 (page 81), la sémantique d'exécution d'une application ATHAPASCAN est basée sur les contraintes de précédence induites par le graphe de flot de données. Ceci permet de définir un invariant de l'état pour les applications ATHAPASCAN qui est indépendant d'une application donnée.

Ainsi, nous définissons l'**invariant de sémantique Athapascan** qui signifie que la sémantique ATHAPASCAN de l'application ne doit pas être modifiée. Cet invariant s'exprime en utilisant l'ordre d'exécution des tâches induit par le graphe de flot de données de l'application : l'ordre d'exécution des tâches après la reconfiguration doit être compatible avec l'ordre d'exécution des tâches avant la reconfiguration.

Notations

- Soit une reconfiguration R.
- Soit G (respectivement G') le graphe de flot de données représentant l'état de l'application avant (respectivement après) la reconfiguration R.
- Soit $\prec_G$ (respectivement $\prec_{G'}$) l'ordre partiel d'exécution des tâches défini par le graphe G (respectivement G').

Définition 9 *On dit que la reconfiguration R préserve l'**invariant de sémantique Athapascan** d'une application si l'ordre $\prec_{G'}$ est compatible avec l'ordre $\prec_G$, c'est-à-dire si pour tout couple d'éléments (a, b), $(a \prec_G b) \Rightarrow (a \prec_{G'} b)$.*

Les reconfigurations comme le vol de travail ou le partitionnement statique conservent l'invariant de sémantique ATHAPASCAN quelle que soit l'application considérée. Nous l'avons montré dans les sections 4.3.2.2 et 4.3.2.3.

Nous les désignons comme les reconfigurations indépendantes de l'application. Inversement, d'autres reconfigurations ne préservent cet invariant que sous certaines hypothèses qui sont liées à l'application. Ces reconfigurations ne sont donc valides que pour une application donnée. Nous les désignons comme les reconfigurations dépendantes de l'application.

5.4.3 Cohérence mutuelle

Des problèmes de cohérence entre deux objets peuvent apparaitre à partir du moment où ils communiquent, c'est-à-dire qu'ils échangent des informations. Nous considérons alors que deux objets sont dans un état mutuellement cohérent si les deux objets ont la même vision de toutes les communications qu'ils ont échangées, c'est-à-dire que tout message émis par un objet apparait reçu par l'autre objet et inversement. Pour cela, il faut noter que cette partie de l'état doit être accessible, c'est-à-dire que cette information doit apparaitre dans la structure de données représentant l'application.

Telle qu'elle est définie, la notion de cohérence mutuelle n'a de sens que pour deux objets qui sont potentiellement modifiés par deux flots d'exécution différents. Cela correspond aux reconfigurations que nous avons qualifiées de distribuées dans la section 5.1.1.

En effet, comme nous l'avons expliqué dans la section 2.4.1 du chapitre 2 (page 32), l'état d'une application est constitué deux éléments : l'état local des processus et l'état des canaux de communication. L'état local d'un processus est accessible localement sur chaque processus tandis que l'état des canaux de communication n'est pas accessible directement. Ainsi, la cause de l'incohérence d'un état observé est due au fait que l'état observé est incomplet puisqu'il manque les informations liées aux messages en transit.

Dans le cas de la sauvegarde coordonnée pour la tolérance aux fautes, l'état des canaux de communication n'a pas besoin d'être accessible directement, mais il suffit qu'il puisse être identifié. C'est par exemple le cas de la sauvegarde coordonnée non bloquante de Chandy et Lamport qui se déroule en deux temps [49] : d'abord l'état local du processus est sauvegardé, puis les messages des canaux qui sont reçus depuis la sauvegarde locale et jusqu'à la reception du « message marqueur » sont sauvegardés. Ces « messages marqueurs » sont chargés de pousser les messages en transit. Cette méthode permet de faire remonter l'état des canaux au niveau applicatif. Mais il faut remarquer que l'état du processus et l'état des canaux de communication ne sont pas accessibles simultanément. C'est la reconstruction de l'état à la reprise qui permet d'avoir l'état cohérent.

Dans le cas de la reconfiguration, il est nécessaire de pouvoir, à un instant donné, accéder à un état cohérent (sans devoir le reconstruire comme dans une reprise). D'où la nécessité de prendre en compte les deux aspects **cohérence** et **accessibilité** de l'état. Le problème de la cohérence mutuelle pour la reconfiguration s'apparente donc plus au cas de la sauvegarde coordonnée bloquante, pour laquelle la sauvegarde ne s'effectue qu'en un temps, puisque l'on attend que l'état des canaux de communication soit nul avant de sauvegarder l'état local d'un processus.

Nous appelons **point local de reconfiguration** le moment, dans l'exécution d'un processus de l'application, où l'état local du processus est observé pour effectuer la reconfiguration. Durant toute la durée de la reconfiguration, les flots d'exécution du processus sont arrêtés. L'état local du processus n'est donc pas modifié autrement que par l'opération de reconfiguration. Le point local de reconfiguration apparait donc comme ponctuel du point de vue des flots d'exécution. De même, le **point global de reconfiguration** désigne l'ensemble des points locaux de reconfiguration de tous les processus participant à la reconfiguration. Sur une échelle de temps absolu, les

points locaux qui composent le point global de reconfiguration ne s'exécutent pas simultanément. Cependant, ce point global de reconfiguration apparait comme ponctuel du point de vue de l'exécution de l'application car les processus ne possèdent pas d'horloge globale.

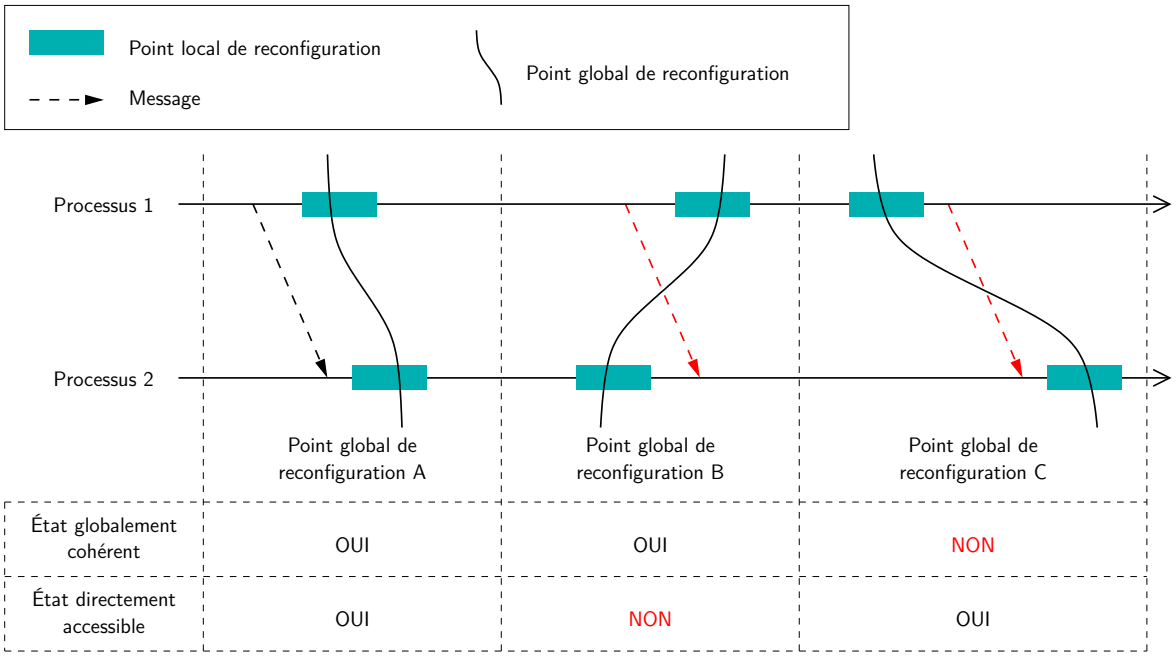


FIG. 5.8: Cohérence et accessibilité des points globaux de reconfiguration

La figure 5.8 montre plusieurs exemples de coupe de l'état qui représentent des points globaux de reconfiguration. Le point global de reconfiguration C n'est pas cohérent puisqu'il représente un état qui ne peut pas se produire dans une exécution normale (le message apparait comme reçu mais non émis). Le point global de reconfiguration B est cohérent (il représente un état qui peut se produire lors d'une exécution normale), cependant l'état des canaux de communication n'est pas vide (un message traverse la ligne de coupe du point global). Enfin, le point global A représente un état à la fois cohérent et accessible directement. Il peut donc être utilisé pour appliquer une reconfiguration qui requiert une cohérence mutuelle entre les objets visés.

Le problème de la cohérence mutuelle d'un ensemble d'objets est similaire à celui de la sauvegarde coordonnée bloquante pour la tolérance aux fautes. La différence ici est que nous ne nous intéressons pas forcément à l'application tout entière, mais éventuellement à un sous-ensemble d'objets qui composent l'application. Donc, pour observer un ensemble d'objets dans un état mutuellement cohérent, il faut rendre accessible l'état des canaux de communication entre ces objets. Pour rendre accessible l'état d'un canal de communication, la sauvegarde coordonnée bloquante vide ce canal de communication à l'aide d'un « message marqueur » spécifique chargé de pousser les messages en transit. Ceci nécessite, bien entendu, que les canaux de communication soient FIFO.

Remarquons que rendre l'état des canaux de communication accessible ne se limite pas à vider les canaux de communication. Il faut également que l'émission et la réception

du message apparaissent dans ce qui est considéré comme l'état de l'application. Par exemple, si l'état de l'application est considéré comme l'espace mémoire utilisateur alloué par l'application, il faut inscrire l'émission et la réception des messages dans l'espace mémoire utilisateur (puisqu'on n'inclut pas les informations du noyau dans l'état de l'application). Dans le cas du moteur d'exécution KAAPI où l'état est représenté par un graphe de flot de données, l'état des émissions et des réceptions de messages est accessible dans le graphe sous forme de tâches et d'attributs d'état sur les tâches.

5.4.3.1 Formalisation

Nous considérons le système composé par les processus qui contiennent les objets visés par la reconfiguration et les canaux de communication entre ces processus.

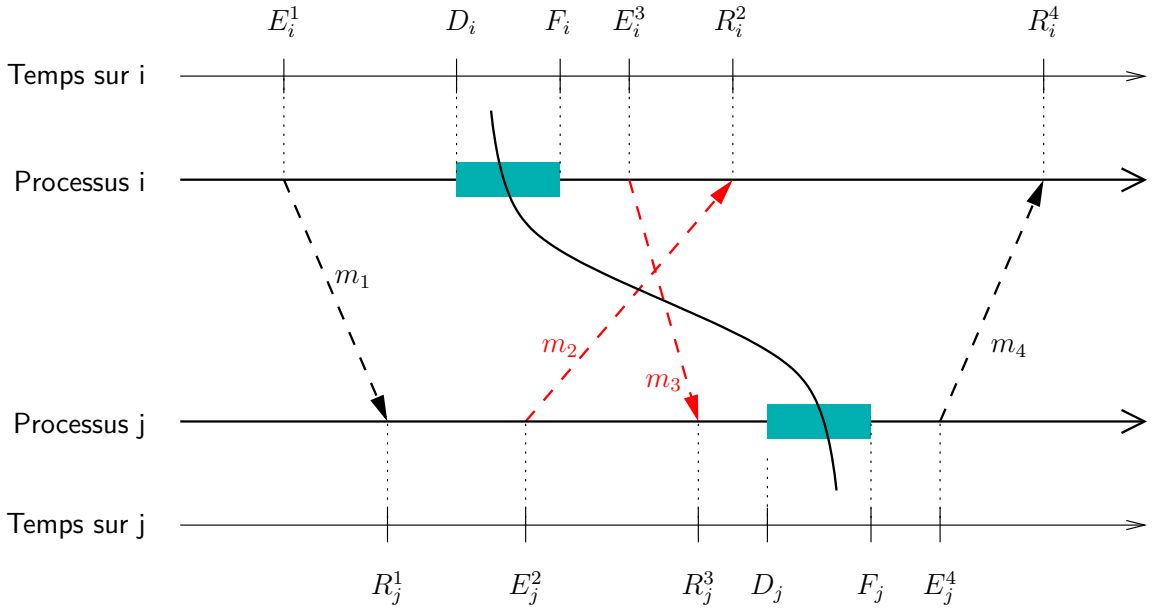


FIG. 5.9: Modèle pour la gestion de la cohérence mutuelle d'une reconfiguration

Notations Pour une reconfiguration donnée et pour le point global de reconfiguration PG choisi, nous utilisons les notations suivantes.

- Le point local de reconfiguration sur le processus P_i est noté PL_i.
- Les dates D_i et F_i correspondent respectivement à la date de début et de fin du point local de reconfiguration du processus P_i.
- Pour tout message m_k de P_i vers P_j, nous notons E_i^k la date d'émission du message m_k sur le processus P_i et R_j^k la date de réception du message m_k sur le processus P_j.

Ces notations sont illustrées sur la figure 5.9.

Hypothèses. Nous ajoutons deux hypothèses supplémentaires à notre problème.

- Tout d'abord, nous précisons que le début d'un point local de reconfiguration précède toujours la fin du point local de reconfiguration. Pour tout processus P_i,

$$D_i < F_i$$

- De plus, durant l'exécution d'une reconfiguration, la cible est isolée. C'est-à-dire qu'elle ne reçoit et n'émet aucun message. Nous l'exprimons formellement de la manière suivante. Pour tout message m_k de P_i vers P_j,

$$E_i^k < D_i \vee F_i < E_i^k \text{ et } R_j^k < D_j \vee F_j < R_j^k$$

  Nous appelons cette propriété l'**isolation** des points locaux de reconfiguration. L'isolation permet de considérer que le point de reconfiguration est ponctuel vis-à-vis des communications puisque toute communication est traitée soit avant, soit après le point de reconfiguration.

Comme nous l'avons dit à la définition 1 page 32, selon Chandy et Lamport, un état cohérent est un état dans lequel, si l'état d'un processus reflète la réception d'un message, alors l'état du processus qui a émis le message doit refléter l'émission du message [49]. Nous définissons donc la propriété de cohérence globale dans notre formalisme.

Définition 10 *L'état d'un point global de reconfiguration est* ***globalement cohérent*** *selon Chandy et Lamport [49] si et seulement si pour tout message m_k de P_i vers P_j,*

$$R_j^k < D_j \Rightarrow E_i^k < D_i$$

De même, nous définissons la propriété d'accessibilité de l'état qui doit garantir qu'aucun message n'est présent dans les canaux de communication dans la coupe d'état correspondant au point global de reconfiguration. Nous l'exprimons en disant que si un message a été émis avant le point local sur le processus émetteur, alors il doit être reçu avant le point local sur le processus récepteur.

Définition 11 *L'état d'un point global de reconfiguration est* ***accessible*** *si et seulement si pour tout message m_k de P_i vers P_j,*

$$E_i^k < D_i \Rightarrow R_j^k < D_j$$

Enfin, nous définissons la cohérence mutuelle qui doit assurer que tous les processus ont la même vision de toutes les communications qu'ils ont échangées.

Définition 12 *L'état d'un point global de reconfiguration est* ***mutuellement cohérent*** *si et seulement si pour tout message m_k de P_i vers P_j,*

$$E_i^k < D_i \Leftrightarrow R_j^k < D_j \text{ et } F_i < E_i^k \Leftrightarrow F_j < R_j^k$$

La proposition suivante se montre alors trivialement à partir des définitions 10, 11, 12 et de l'hypothèse d'isolation.

Proposition 5 *Sous l'hypothèse d'isolation, l'état d'un point global de reconfiguration est mutuellement cohérent si et seulement si cet état est globalement cohérent et accessible.*

5.4.3.2 Contraintes de cohérence mutuelle

D'après la section précédente, pour construire un protocole de gestion de la cohérence mutuelle, il est nécessaire et suffisant de vérifier les trois contraintes : isolation, cohérence et accessibilité.

Isolation. Elle doit garantir que pendant la phase de mutation, les objets visés par la reconfiguration ne seront pas modifiés, excepté par le flot de reconfiguration. Elle doit être assuré vis-à-vis des flots de d'exécution et également des communications (que ce soit les communications en provenance des autres objets visés par la reconfiguration, ou bien celles en provenance d'objets de l'application qui ne participent pas à la reconfiguration).

L'isolation est réalisée grâce aux méthodes de gestion des accès concurrents qui permettent d'éviter qu'un objet en cours de mutation soit modifié par un autre flot que le flot de reconfiguration qui effectue la mutation.

Accessibilité. D'après la définition 11, l'accessibilité signifie que si un message a été émis avant le point local de reconfiguration sur un autre processus, alors le point local de reconfiguration doit être après la réception de ce message sur le processus récepteur. Concrètement, cela signifie qu'avant d'effectuer la mutation (qui correspond au point local), il faut s'assurer que les canaux de communication ne contiennent pas de messages qui ont été émis avant un point local sur un autre processus. L'accessibilité est donc garantie en utilisant des techniques de vidage des canaux de communication (supposés FIFO).

L'opération de vidage peut être réalisée de plusieurs manières selon le type du canal de communication. Généralement, les canaux considérés sont de réseaux de communication entre des machines distantes qui reposent sur des protocoles FIFO (*First In First Out*). Dans ce cas, l'opération de vidage du canal de communication est effectuée en émettant un « message marqueur » à travers le canal de communication. Ce message marqueur est le dernier message émis sur le canal avant le point local de reconfiguration. À la réception de ce message marqueur, grâce à la propriété FIFO du canal et si aucun autre message n'a été émis après le message marqueur, alors le canal de communication est vide.

Sans connaissance particulière sur le schéma de communication de l'application, le vidage des canaux de communication entre n processus est réalisé classiquement de la manière suivante. Chacun des n processus émet à chacun des $n-1$ autres processus le message marqueur. Lorsqu'un processus a reçu $n-1$ messages marqueurs, alors il est sûr que tous les canaux de communication qui lui délivrent des messages sont vides. Cette méthode requiert l'émission de $O(n^2)$ messages marqueurs pour vider les canaux de communication entre n processus.

Cohérence. Le but de la cohérence est d'empêcher que le point global constitue un état impossible à l'exécution vis-à-vis des messages (puisque les points locaux ne peuvent pas s'exécuter exactement en même temps), c'est-à-dire un état inatteignable dans une exécution normale. Grâce à l'hypothèse d'isolation, la définition 10 de la cohérence globale est équivalente à la proposition suivante.

Pour tout message m_k de P_i vers P_j,

$$F_i < E_i^k \Rightarrow F_j < R_j^k$$

Nous exprimons cette proposition en disant que pour garantir la cohérence, il faut que tout message émis après le point local sur un processus soit reçu après le point local sur le processus récepteur.

Deux approches sont possibles pour vérifier cette contrainte : bloquer les messages à l'émission ou à la réception.

- Bloquer les messages à l'émission consiste à bloquer l'exécution des objets cibles tant que tous les objets cibles n'ont pas fini leur point local de reconfiguration.
- Bloquer les messages à la réception consiste à ne pas délivrer les messages des canaux de communication à partir du moment où le message marqueur à été reçu et tant que le point local de reconfiguration n'est pas terminé. Ceci est réalisé sur le processus récepteur en conservant ces messages en mémoire sans les délivrer à l'application.

5.4.3.3 Cohérence mutuelle dans Kaapi

Nous proposons pour Kaapi, une interface de gestion de la cohérence qui permet au programmeur d'une reconfiguration de demander un état mutuellement cohérent entre l'ensemble des objets cibles de la reconfiguration. Le protocole de coordination est alors réalisé par le moteur d'exécution Kaapi. Cette interface se compose des deux appels suivants.

- `acquire_mutual_consistency()` indique l'entrée dans le point local de reconfiguration d'un processus. C'est un appel bloquant dont le but est de coordonner les processus qui participent à la reconfiguration pour garantir un état mutuellement cohérent entre les objets cibles de la reconfiguration. Nous rappelons que l'étape de réalisation d'une reconfiguration se comporte comme un programme SPMD. Cet appel doit être réalisé dans chaque flot de reconfiguration d'une même reconfiguration pendant l'étape de réalisation. Au moment de l'appel, l'ensemble des processus impliqués est déjà connu puisqu'il s'agit de la cible de la reconfiguration. Sur un processus donné, l'appel bloque le flot de reconfiguration jusqu'à ce que la cohérence mutuelle soit assurée sur ce processus.
- `release_mutual_consistency()` indique la sortie du point local de reconfiguration d'un processus. C'est un appel bloquant qui permet d'indiquer que la reconfiguration est terminée, ou au moins que les modifications nécessitant un état mutuellement cohérent sont terminées.

Cette interface permet au programmeur d'une reconfiguration de spécifier, s'il en a besoin, la contrainte de cohérence mutuelle.

Le protocole de gestion de la cohérence de Kaapi est déclenché lors de l'appel à la fonction `acquire_mutual_consistency()`. Nous proposons dans Kaapi un protocole de gestion de la cohérence mutuelle qui se base sur la connaissance de l'application pour coordonner les processus cibles.

Le protocole implementé dans Kaapi se déroule en deux étapes. La première étape assure l'accessibilité de l'état en arrêtant les flots d'exécution et en vidant les canaux

de communication. La seconde étape assure la cohérence en bloquant les messages à l'émission. Quant à l'isolation, elle est gérée par les techniques décrites dans la section 5.3.

Réalisation de l'accessibilité. Dans KAAPI, l'accessibilité est réalisée en vidant les canaux de communication. Pour vider les canaux de communication, nous proposons une méthode qui réduit le nombre de messages émis par rapport à la méthode classique où chaque processus émet un message à tous les autres processus. Notre méthode est basée sur la connaissance que nous avons de l'application grâce au graphe de flot de données qui représente le futur de l'exécution. Le principe est de determiner, grâce aux informations contenues dans le graphe de flot de données, les communications qui sont potentiellement en cours et ainsi de ne vider que les canaux de communication potentiellement non vides.

Étant donné que la cohérence mutuelle se fait vis-à-vis de l'état de l'application, nous ne prenons en compte que les communications qui influencent l'état de l'application. Ainsi tous les messages, comme les messages de contrôle utilisés par le moteur d'exécution KAAPI ne sont pas concernés.

Les informations sur les communications sont présentées dans le graphe de flot de données de plusieurs manières.

- L'émission de données apparait sous la forme d'une tâche de communication `Broadcast` sur l'émetteur et par une tâche de communication `Receive` sur le récepteur.
- Un vol de travail apparait sous forme d'un attribut d'état *Volée* sur la tâche volée sur la victime et par un attribut de vol sur le K-processeur sur le processus voleur.
- Un retour de vol (*i.e.* l'émission du résultat après l'exécution d'une tâche volée) est identifié par une tâche `Signal` chargée de retourner le résultat sur le voleur et par la tâche dans l'état *volée* sur la victime.

Ces communications sont donc les seules qui sont prises en compte dans ce protocole optimisé.

Nous devons également remarquer que la representation abstraite de l'application dans KAAPI nous permet d'accéder au futur de l'exécution, mais pas au passé de l'exécution car une fois exécutée par KAAPI, la tâche est détruite pour libérer la mémoire. Cela signifie qu'il n'est pas possible de savoir si un message vient d'être émis. Par contre, il est possible d'identifier ceux qui n'ont pas encore été reçus.

Sur un processus P_i, l'exécution de l'appel `acquire_mutual_consistency()` provoque l'arrêt des flots d'exécution du processus. Ensuite, l'analyse du graphe de flot de données du processus P_i permet de déterminer l'ensemble des communications qui peuvent être reçues et donc l'ensemble des émetteurs potentiels, noté E. Ce sont les canaux de communication entre les émetteurs potentiels et ce processus qui doivent être vidés.

Pour cela, le vidage des canaux de communication est fait par un aller-retour de messages (PING-PONG). Si le processus P_i a comme émetteur potentiel le processus P_j, alors P_i émet le message PING à P_j. Quand le processus P_j reçoit le message PING en provenance de P_i, cela lui indique que P_i est un de ses processus récepteurs.

Sur réception d'un message PING et si le processus P_j a arrêté ses flots d'exécution alors il émet en retour le message PONG à P_i. S'il n'a pas arrêté ses flots d'exécution,

P_j enregistre le message et le message PONG ne sera émis vers P_i qu'après l'arrêt des flots d'exécution. Ceci permet de garantir que plus aucun message lié au calcul ne sera émis après le message PONG par l'exécution de tâches de communication. À la réception du message PONG en provenance de P_j, le processus P_i est assuré que le processus P_j est arrêté (donc il n'enverra plus de nouveau message) et que les messages émis précédemment ont été reçus. Le message PONG joue ici le rôle du « message marqueur ». L'appel à la fonction `acquire_mutual_consistency()` se termine lorsque tous les canaux sont vides.

Le nombre de messages échangés à cette étape dépend du nombre de voisins de chaque processus. Si nous notons n le nombre de processus participant à la reconfiguration et v le nombre moyen de voisins des processus, alors le nombre de messages émis pour le vidage des canaux de communication est en $O(n \times v)$.

Réalisation de la cohérence. Dans l'implémentation actuelle de Kaapi, la cohérence est assurée en bloquant les messages à l'émission. Plus précisément, le principe est de bloquer, jusqu'à la fin du point local de reconfiguration, les flots d'exécution distants qui peuvent générer l'émission de messages . Ceci est réalisé de la manière suivante.

Les flots d'exécution des processus cibles ont été arrêtés au début de l'appel à `acquire_mutual_consistency()`. Nous réalisons la cohérence grâce à un bloquage à l'émission, c'est-à-dire que chaque processus ne doit pas émettre de message vers un processus qui n'a pas fini son point local de reconfiguration. Pour cela, nous utilisons un message CONT qui permet à un processus de signaler à ses voisins (*i.e.* l'ensemble des émetteurs potentiels calculé lors de l'étape de vidage de canaux de communication) qu'il est sorti de son point local de reconfiguration.

Pour indiquer la fin du point local de reconfiguration, le flot de reconfiguration appelle la fonction `release_mutual_consistency()`. Avant de redémarrer ses flots d'exécution, le processus attend de recevoir tous les messages CONT ; il doit en recevoir autant qu'il a reçu de messages PING. Ensuite, il peut redémarrer les flots d'exécution et sortir de l'appel à la fonction `release_mutual_consistency()`.

Pour vérifier la validité de ce protocole, nous nous basons sur les points suivants.

- La réception d'un message CONT signifie que l'émetteur a fini son point local de reconfiguration.
- Avant de redémarrer ses flots d'exécution, un processus attend de recevoir un message CONT de la part de tous les processus vers lesquels il peut émettre des messages.
- Un processus ne peut émettre des messages applicatifs qu'après avoir redémarré ses flots d'exécution.

Ce protocole garantit donc qu'aucun message émis après un point local de reconfiguration sur un processus ne sera reçu avant la fin d'un point local de reconfiguration sur un autre processus. Il permet donc d'assurer la cohérence globale.

La figure 5.10 montre l'algorithme complet qui comprend le vidage des canaux de communication et le bloquage des messages à l'émission. Le protocole est illustré sur la figure 5.11. Sur cette figure, la reconfiguration s'applique sur les processus 1, 2 et 3. Le processus 1 peut recevoir des communications en provenance de 2 uniquement ; le processus 2 peut recevoir des communications de 1 et 3 ; tandis que le processus 3 peut recevoir des communications de 2 uniquement.

A – Fonction `acquire_mutual_consistency()` :
A.1 – Arrêt des calculs
A.1.1 – Arrêt des flots d'exécution
A.1.2 – Réémettre un message PONG à P_j pour chaque message PING reçu de P_j et mis en attente en C.3
A.2 – Vidage des canaux de communication
A.2.1 – Initialisation de N_{PING} , le compteur de messages reçus, à 0
A.2.2 – Calcul de $E = \{\text{processus émetteurs potentiels}\}$
A.2.3 – $\forall k \in E$, émission du message PING à P_k
A.2.4 – $\forall k \in E$, attente du message PONG de P_k

B – Fonction `release_mutual_consistency()` :
B.1 – Signalisation de la fin du point local de reconfiguration
B.1.1 – $\forall k \in E$, émission du message CONT à P_k
B.1.2 – Attente de N_{PING} messages CONT
B.2 – Redémarrage des flots d'exécution

C – Sur réception d'un message PING :
C.1 – Incrémenter N_{PING}
C.2 – Si les calculs sont arrêtés, réémettre PONG vers l'émetteur
C.3 – Sinon mettre le message en attente jusqu'à l'exécution de l'étape A.1.2

D – Sur réception d'un message PONG :
D.1 – Si le nombre de messages PONG reçus est $Card(E)$, alors réveiller le processus bloqué en 1.2.4

E – Sur réception d'un message CONT :
E.1 – Si le nombre de messages CONT reçus est N_{PING}, alors réveiller le processus bloqué en B.1.2

FIG. 5.10: Algorithme de cohérence mutuelle sur les processus cibles

5.5 Conclusion

Dans ce chapitre, nous avons présenté des mécanismes pour faciliter la reconfiguration d'applications parallèles distribuées. Ce chapitre a abordé ce problème à travers la gestion des accès concurrents et la gestion de la cohérence.

La gestion des accès concurrents est un aspect peu étudié dans la littérature du point de vue général des reconfigurations. Notre contribution a porté sur une modélisation des problèmes de concurrence. Nous avons proposé deux méthodes d'exécution. L'exécution concurrente permet d'exécuter une reconfiguration en concurrence des flots d'exécution si elle cible des objets inactifs. L'exécution coopérative préempte un flot d'exécution et délègue l'exécution de la reconfiguration au processus léger qui exécutait le flot d'exécution de l'application. Cette dernière approche permet une implémentation sans primitive de synchronisation.

La gestion de la cohérence, en particulier la gestion de la cohérence mutuelle, est un problème largement abordé dans la littérature. Pour résoudre ce problème, nous

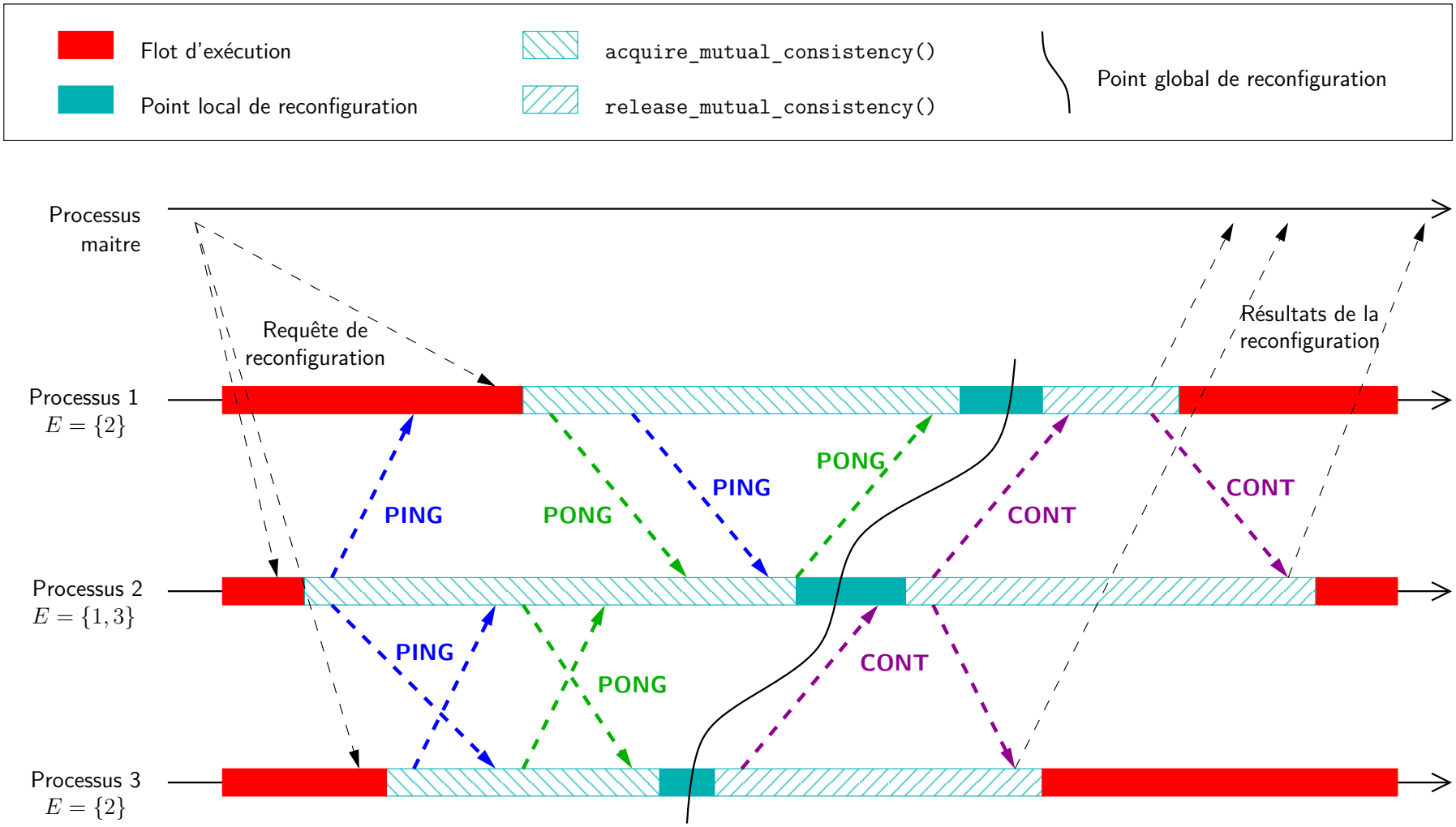


FIG. 5.11: Protocole de cohérence mutuelle dans KAAPI

avons fait le rapprochement avec les protocoles de tolérance aux fautes par sauvegarde coordonnée bloquante ou non bloquante qui utilisent la notion de cohérence definie par Chandy et Lamport. Nous avons défini une propriété supplémentaire, l'accessibilité, qui associée à la cohérence selon Chandy et Lamport, permet de garantir la cohérence mutuelle d'un ensemble de processus pour la reconfiguration.

Enfin, nous avons proposé pour chacun de ces aspects une implémentation dans le moteur d'exécution Kaapi.

Expérimentations 6

Sommaire

6.1 Introduction

Dans ce chapitre, nous nous intéressons à évaluer le cout de réalisation des mécanismes de reconfiguration que nous avons proposés au chapitre précédent.

Dans la section 6.2 de ce chapitre, nous effectuons la comparaison entre l'exécution concurrente et l'exécution coopérative d'une reconfiguration. Pour cela, nous nous attacherons uniquement à la reconfiguration « vol de travail » à travers le logiciel X-Kaapi.

Dans la section 6.3, nous mesurons le cout du mécanisme permettant de garantir la cohérence mutuelle dans Kaapi. Cette brique de base est importante. Elle est notamment utilisée avec les protocoles de tolérance aux fautes qui seront vus dans la partie suivante.

6.2 Vol de travail concurrent et coopératif

Nous comparons tout d'abord les implémentations concurrente et coopérative du vol de travail sur une application simple. Ensuite, nous utilisons l'implémentation coopérative du vol de travail pour se comparer à Cilk et TBB.

6.2.1 Comparaison des deux approches

Nous comparons deux implémentations du vol de travail au sein du logiciel X-Kaapi. La première est basée sur le principe de l'exécution concurrente présentée dans la

section 5.3.3 ; la seconde utilise la méthode d'exécution coopérative exposée dans la section 5.3.4.

L'opération de vol de travail pour le modèle de programmation Kaapi est présentée dans la section 4.3.2.2. Le logiciel X-Kaapi ne fonctionne (pour le moment) qu'en mémoire partagée ; le vol de travail proposé n'opère donc qu'entre des objets locaux. La reconfiguration « vol de travail » est déclenchée lorsqu'un K-processeur devient inactif. Dans ces implémentations, la cible du vol de travail est un K-thread, appelé victime ; l'unique R-objet modifié (ou marqué) est une K-task, dénommée comme la tâche volée.

Il faut remarquer que l'implémentation de la reconfiguration « vol de travail » reste la même dans les deux cas. Seule l'implémentation du mécanisme de reconfiguration change entre la version concurrente et la version coopérative. Comme cela a été expliqué dans les sections 5.3.3 et 5.3.4, la version concurrente utilise des primitives de synchronisation (verrou, instruction `compare-and-swap`) pour garantir l'exclusion mutuelle entre le flot de reconfiguration et le flot d'exécution qui sont chacun exécutés par un processus léger qui accèdent de manière concurrente aux K-threads. Quant à la version coopérative, elle permet au processus léger qui exécute le flot d'exécution de s'interrompre pour exécuter le flot de reconfiguration, ce qui nécessite un certain délai.

L'objectif de ces premières expériences est de mesurer le cout de ces deux approches sur un problème simple, le calcul du N$^{\text{e}}$ terme de la suite de Fibonacci. Cette application est une implémentation naïve de l'algorithme récursif de la suite de Fibonnaci, similaire à celle présentée à la figure 4.2b.

Cette application nous permet pour une même instance, c'est-à-dire pour N fixé, de faire varier facilement le nombre de tâches créées. Ceci est réalisé en modifiant le seuil, noté s, en dessous duquel une tâche de calcul calcule son résultat séquentiellement au lieu de créer de nouvelles tâches pour le calculer en parallèle.

Un seuil s petit correspond à la création de beaucoup de tâches à grain fin ; tandis qu'un seuil s grand implique la création de peu de tâches à gros grain. Grâce à cela, nous pouvons observer le cout induit par la gestion des tâches des mécanismes d'exécution concurrent et coopératif.

Toutes les expériences de cette section ont été réalisées sur une machine à 8 cœurs de la grappe Digitalis de Grenoble de Grid'5000. Les temps reportés sont la moyenne d'au moins 10 mesures. Les écarts types des mesures sont généralement très faibles, de l'ordre de 1 %.

6.2.1.1 À grain fin

Dans cette expérience, nous cherchons essentiellement à mesurer le cout induit pour une application en créant un grand nombre de tâches à grain fin : le calcul du N$^{\text{e}}$ nombre de Fibonacci s'y prête particulièrement bien car les instructions arithmétiques exécutées par les tâches sont très faibles.

Pour l'instance considérée, $N = 45$, le temps moyen d'exécution du programme séquentiel est 7,197 secondes. Les deux versions parallèles, concurrente et coopérative, utilisent un seuil d'arrêt de découpe $s = 5$ en deçà duquel le code séquentiel est exécuté[1]. Avec ce seuil, le nombre total de tâches exécutées est 3 467 955 496.

[1] Ce seuil d'arrêt est le même que celui pris par Cilk dans [39].

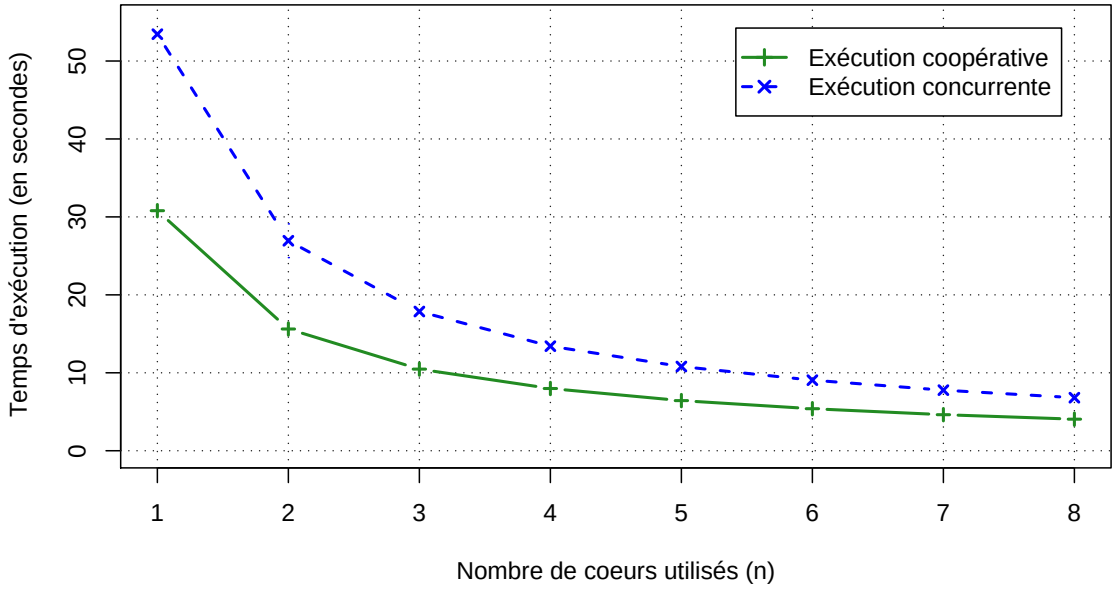


FIG. 6.1: Temps d'exécution de Fibonacci ($N = 45$, $s = 5$) pour les versions concurrente et coopérative de la reconfiguration « vol de travail »

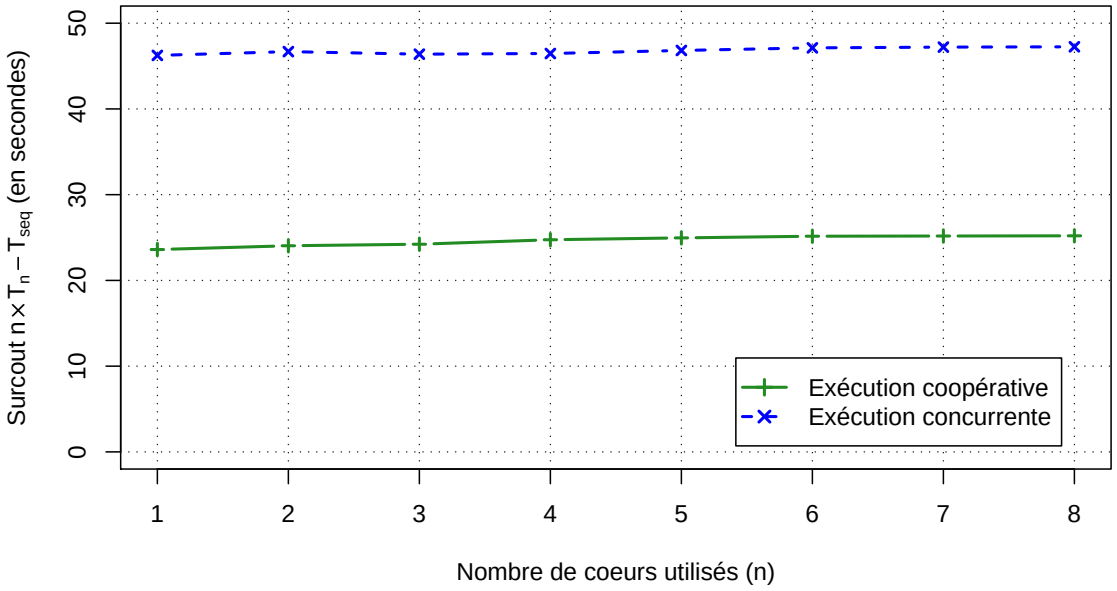


FIG. 6.2: Surcout de gestion du parallélisme $n \times T_n - T_{seq}$ avec Fibonacci à grain fin ($N = 45$, $s = 5$) pour les versions concurrente et coopérative de la reconfiguration « vol de travail »

La figure 6.1 montre les temps obtenus en faisant varier le nombre de cœurs. Le temps sur 1 cœur de la version concurrente est de 53,440 secondes ; cette version est 7,43 fois plus lente que la version séquentielle. Quant à la version coopérative, elle s'exécute en 10,796 secondes sur 1 cœur ; elle est 4,28 fois plus lente que le programme séquentiel.

Le figure 6.2 montre le surcout dû à la gestion des tâches de l'application lors de l'exécution. Ce surcout est calculé de la manière suivante : $(n \times T_n - T_1)$ en prenant n le nombre de cœurs. Le surcout est quasiment constant quel que soit le nombre de cœurs.

Le surcout par tâche $(n \times T_n - T_1)/\#tasks$ est de $1,33 \times 10^{-8}$ seconde pour la version concurrente et de $6,80 \times 10^{-9}$ seconde pour la version coopérative. Bien que ces surcouts soient faibles, du fait du grand nombre de tâches, leur accumulation devient importante.

Cette différence est due à la manière d'exécuter les tâches. L'implémentation concurrente utilise un `compare-and-swap` pour exécuter chaque tâche afin d'éviter les conflits lors des accès concurrents. L'implémentation coopérative garantit qu'il n'y a pas d'accès concurrent sur les tâches, l'utilisation de `compare-and-swap` est donc inutile.

Notons également que ces deux versions possèdent une bonne accélération relative sur 8 cœurs : $7,85$ pour la version concurrente et $7,60$ pour la version coopérative.

6.2.1.2 À gros grain

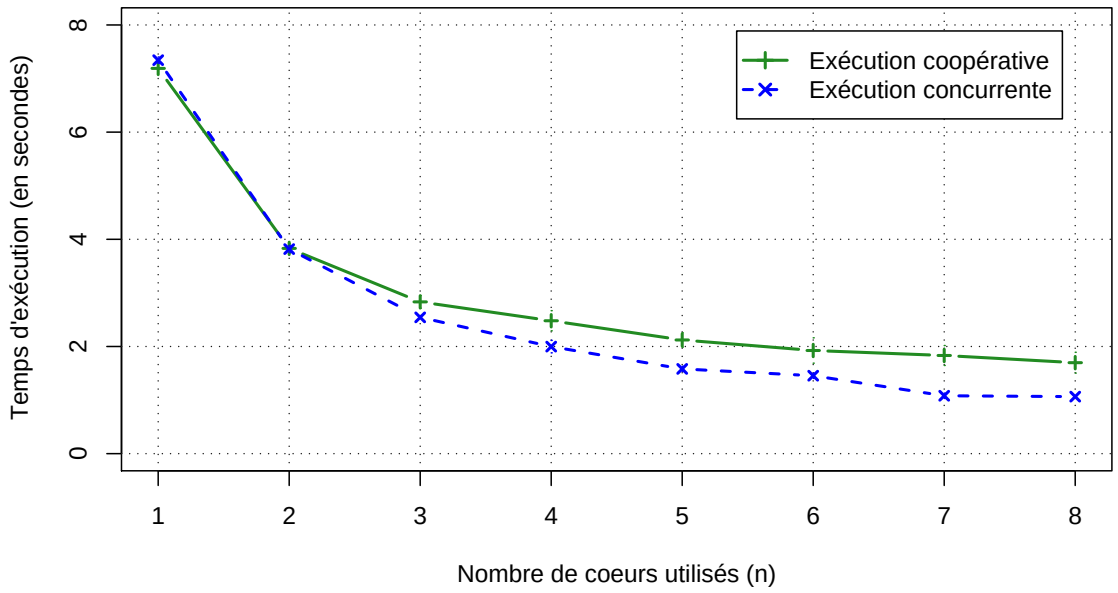


FIG. 6.3: Temps d'exécution de Fibonacci ($N = 45$, $s = 40$) pour les versions concurrente et coopérative de la reconfiguration « vol de travail »

Pour ces nouvelles mesures, nous augmentons le grain de l'application en prenant un seuil d'arrêt s de 40. De ce cas, le nombre de tâches exécutées est 168. Les figures 6.3

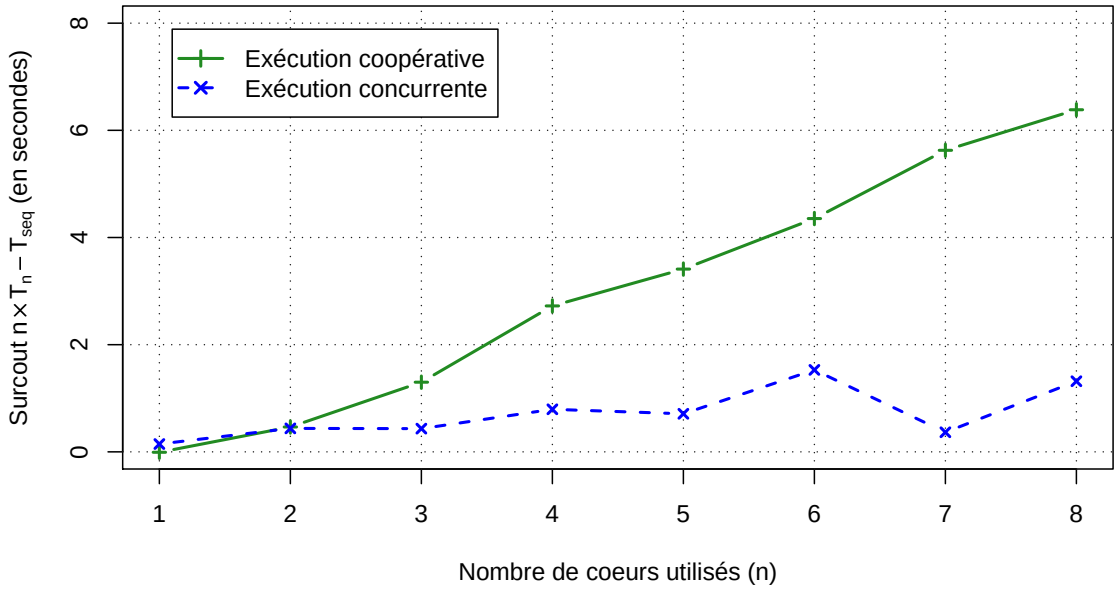


FIG. 6.4: Surcout de gestion du parallélisme $n \times T_n - T_{seq}$ avec Fibonacci à grain fin ($N = 45$, $s = 40$) pour les versions concurrente et coopérative de la reconfiguration « vol de travail »

et 6.4 montrent respectivement le temps d'exécution et le surcout associé pour le calcul de Fibonacci $N = 45$.

Nous remarquons que, à gros grain, la version coopérative est moins efficace que la version concurrente, notamment lorsque le nombre de cœurs utilisés augmente.

En effet, cette version coopérative souffre du manque de réactivité aux requêtes de vols. Le flot d'exécution ne peut coopérer et traiter les requêtes de vol qu'entre l'exécution de deux taches. Lorsque le grain des tâches devient gros, cela affecte le temps de calcul puisqu'il y a un délai important entre le moment de la requête de vol (instanciation de la reconfiguration) et le traitement de la requête (réalisation de la reconfiguration).

La version concurrente du vol de travail ne souffre pas de ce problème puisque, grâce à l'utilisation de primitives de synchronisation, le K-thread voleur (par l'intermédiaire du flot de reconfiguration) peut accéder en concurrence aux tâches du K-thread victime. Dans ce cas, la réponse à la requête de vol est plus rapide. De plus, le surcout lié à l'utilisation des `compare-and-swap` lors de l'exécution de chaque tâche devient négligeable face au temps d'exécution des tâches.

Sur la figure 6.4, nous voyons que le surcout par rapport à l'exécution séquentielle est proportionnel au nombre de processeurs utilisés. En effet, les voleurs sont plus nombreux. Cependant, à cause du gros grain des tâches, le délai de réponse à une requête de vol est important. En conséquence, les processeurs inactifs (et donc voleurs) restent inactifs longtemps.

Cette expérience a mis en évidence le fait qu'un grain trop grand présente des inconvénients.

- Il augmente le chemin critique T_∞ et diminue le degré de parallélisme utilisable dans le cadre des algorithmes à base de vol de travail [39].
- De plus, dans le cas du vol coopératif, le délai de réponse aux requêtes de vol est important.

6.2.2 Comparaison avec Cilk et TBB

Comme le vol de travail est également utilisé dans les bibliothèques Cilk [39] et TBB [99], nous avons voulu comparer leurs performances avec le vol de travail coopératif de X-Kaapi. Pour cette comparaison, nous utilisons les algorithmes `merge`, `min_element` et `transform` de la STL (*Standard Template Library*) sur des tableaux de type `double`. Ces expériences ont été réalisées grâce à l'aide de Daouda Traoré et de Christophe Laferrière et ont abouti à la publication de [31].

L'opération arithmétique effectuée par l'algorithme `transform` est l'opération `*=2`. Les codes sont tous écrits en C++ et compilés avec la version 4.3 du compilateur `g++` avec option d'optimisation `-O2`. La machine d'expérience est une architecture multicœur NUMA composée de 8 processeurs dual-cœurs AMD 875 à 2.2 Ghz. À chaque processeur est associé un banc mémoire de 4 Go, soit 32 Go de mémoire au total. Chaque expérience qui utilise k cœurs ($k = 1$ ou 8) est réalisée en figeant chaque *thread* noyau POSIX sur un des k cœurs utilisés.

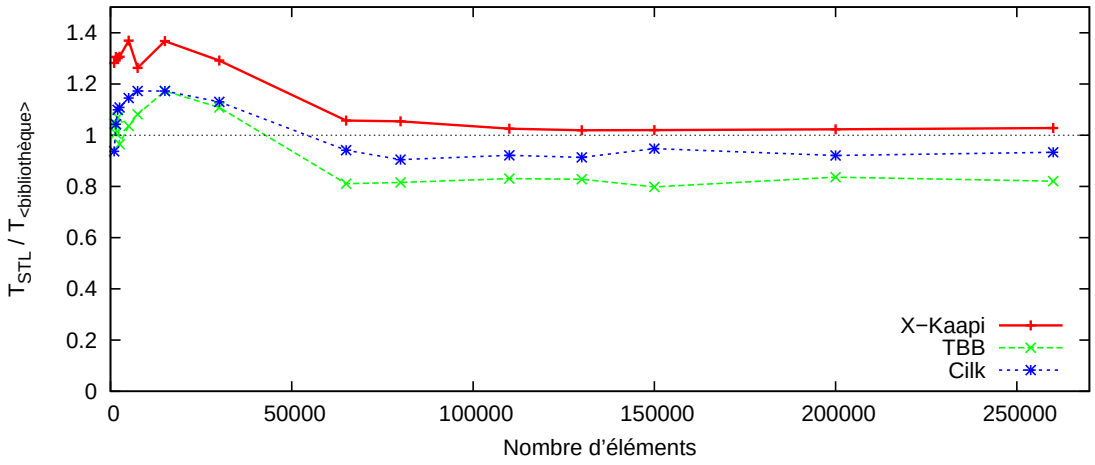
La stratégie d'allocation des tableaux est contrôlée en utilisant `numactl` afin d'allouer chaque page des tableaux sur chacun des k bancs mémoire utilisés de manière cyclique (stratégie `--interleave` de `numactl`).

Chaque valeur tracée correspond à la moyenne des mesures de 30 exécutions du programme avec les mêmes entrées. L'écart type des valeurs est dans tous les cas très petit (de deux ordres de grandeur plus petit que les valeurs). Le programme répète plusieurs fois le même calcul sur les mêmes données. La première mesure est ignorée pour éviter les effets liés aux défauts de cache, l'objectif premier étant de mesurer les couts arithmétiques sans tenir compte des effets de cache.

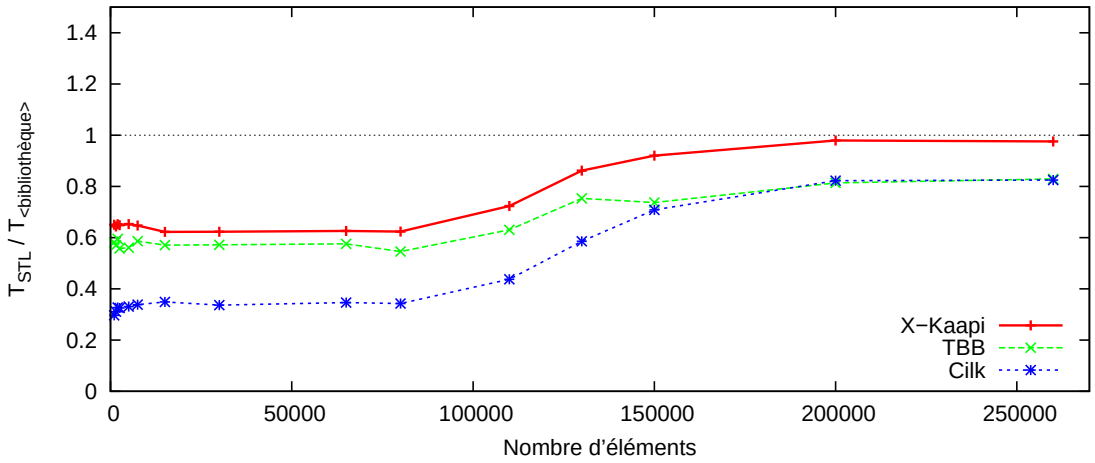
Les figures 6.5 et 6.6 montrent le surcout $T_{seq}/T_{bibliothèque}$ pour chacune des bibliothèques testées en fonction de la taille du tableau d'entrée et pour les trois algorithmes. Le temps séquentiel T_{seq} utilisé comme référence est le temps d'exécution avec la STL C++. La figure 6.5 montre ce surcout lorsqu'un seul cœur est utilisé. La figure 6.6 correspond à l'exécution sur 8 cœurs.

Pour l'exécution sur 8 cœurs, figure 6.6, X-Kaapi permet un gain d'au plus $6,68$ pour `transform` et $6,72$ pour `merge`. Pour `min_element`, le gain maximum observé est $8,95$. Ce gain supérieur à 8 peut s'expliquer par le fait qu'en utilisant 8 cœurs, on bénéficie d'un cache 8 fois plus grand que la STL qui ne tourne que sur un cœur. Dans le cas de `min_element`, le gain représente environ 73% de plus que TBB et 153% de plus que Cilk++.

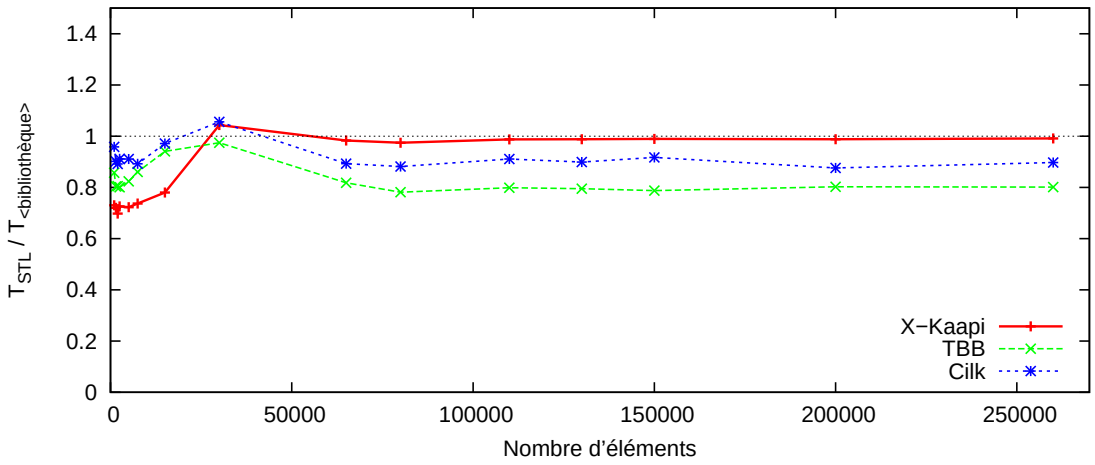
Les temps d'exécution sur 8 cœurs pour 15 000 éléments sont donnés dans le tableau 6.2.2. Le grain est très fin puisque les temps mesurés sont de l'ordre de 10^{-5}



(a) `transform` sur 1 cœur

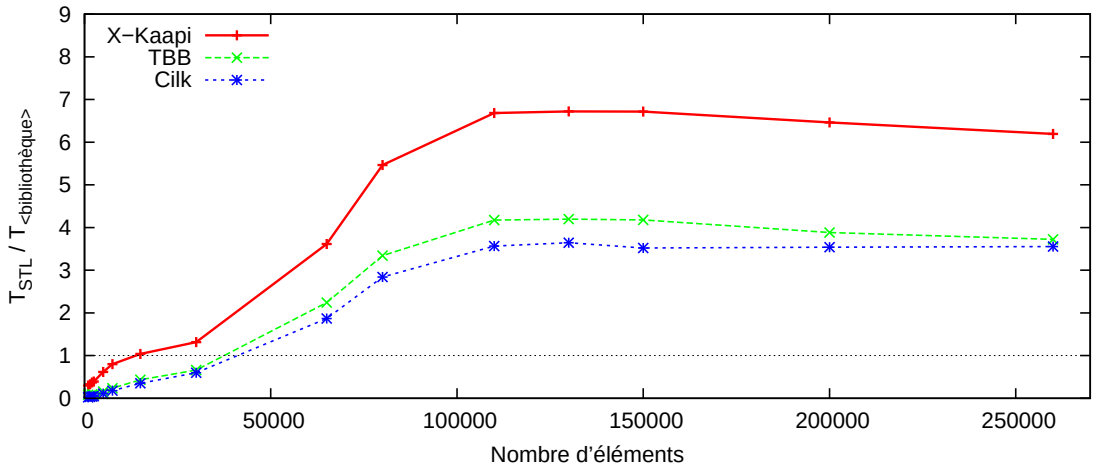


(b) `min_element` sur 1 cœur

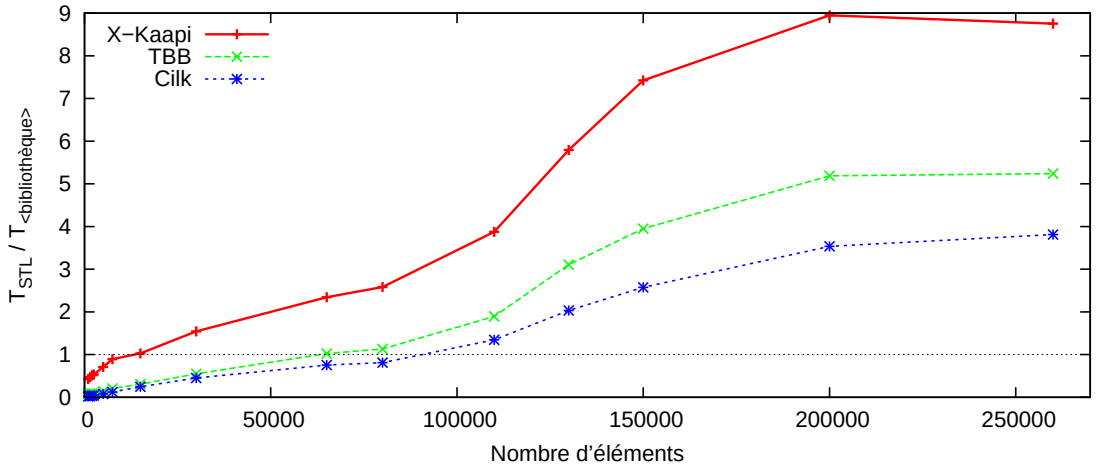


(c) `merge` sur 1 cœur

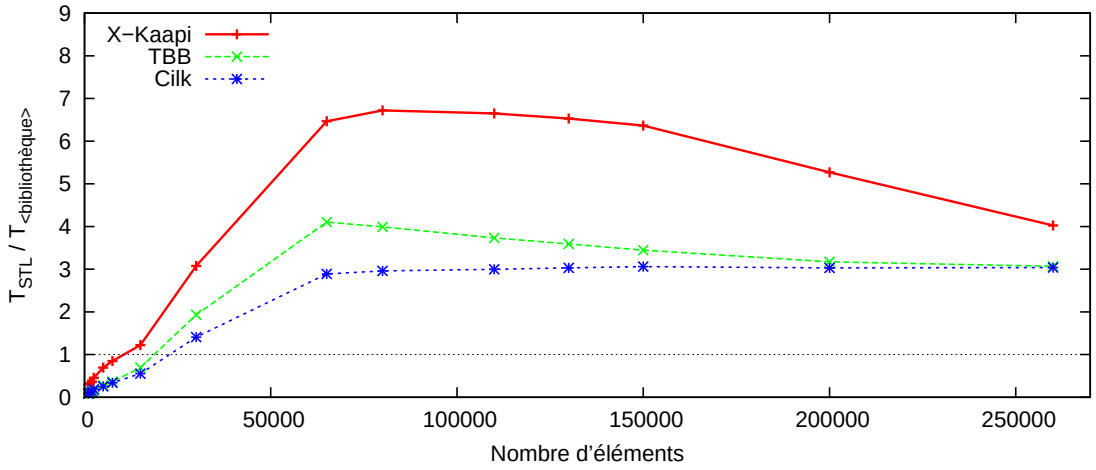
FIG. 6.5: Accélération des bibliothèques par rapport à la STL sur 1 cœur



(a) `transform` sur 8 cœurs



(b) `min_element` sur 8 cœurs



(c) `merge` sur 8 cœurs

FIG. 6.6: Accélération des bibliothèques par rapport à la STL sur 8 cœurs

	X-Kaapi	TBB	Cilk
`transform`	$3,65 \times 10^{-5}$ s	$8,73 \times 10^{-5}$ s	$1,08 \times 10^{-4}$ s
`min_element`	$2,74 \times 10^{-5}$ s	$9,12 \times 10^{-5}$ s	$1,13 \times 10^{-4}$ s
`merge`	$7,15 \times 10^{-5}$ s	$1,26 \times 10^{-4}$ s	$1,58 \times 10^{-4}$ s

Tab. 6.1: Temps d'exécution sur 8 cœurs des algorithmes parallèles `transform`, `min_element` et `merge` pour 15 000 éléments

seconde. Cette taille de 15 000 est la taille à partir de laquelle le code X-Kaapi devient plus rapide que le code séquentiel de la STL sur 8 cœurs. Pour les autres bibliothèques, cette taille est de plus de 30 000.

6.3 Gestion de la cohérence mutuelle

Cette section présente l'évaluation expérimentale du protocole de cohérence mutuelle proposé dans le chapitre précédent. Cette évaluation porte sur deux points. Tout d'abord, nous étudions le nombre de messages échangés par chaque processus pour obtenir la cohérence mutuelle, puis nous mesurons le temps total nécessaire pour réaliser une reconfiguration vide.

Le protocole étudié est le protocole optimisé implémenté dans Kaapi. Comme nous l'avons expliqué dans la section 5.4.3.3, cette version optimisée de la gestion de la cohérence mutuelle utilise le graphe de flot de données de l'application pour déterminer les canaux de communication qui doivent être vidés. Elle sera désignée par la suite comme la **cohérence mutuelle optimisée**.

Durant cette étude expérimentale, nous nous comparons à la version non optimisée de ce protocole de gestion de la cohérence mutuelle. Pour assurer la cohérence mutuelle, cette version, appelée par la suite **cohérence mutuelle non optimisée**, doit vider tous les canaux de communication entre tous les processus puisqu'elle n'utilise pas d'information sur l'état de l'application.

Pour les mesures de la suite de chapitre, nous utilisons une application de résolution du problème des N-reines. Cette application compte le nombre de placements possibles de N dames d'un jeu d'échecs sur un échiquier de $N \times N$ cases, sans qu'elles ne se menacent mutuellement. L'algorithme de résolution est récursif et il est écrit en Athapascan. L'application est exécutée par le moteur d'exécution Kaapi en utilisant un ordonnancement par vol de travail.

6.3.1 Nombre de messages échangés

Nous étudions ici le nombre de messages échangés pour assurer la cohérence mutuelle d'une reconfiguration. Dans le chapitre précédent, nous avons donné la complexité en termes de messages pour les deux versions du protocole.

- La cohérence mutuelle non optimisée nécessite $O(n^2)$ messages.
- La cohérence mutuelle optimisée nécessite $O(n \times v)$ messages.

Nous rappelons les notations suivantes : n désigne le nombre de processus participant au calcul et v le nombre de voisins. Nous désignons ici par le terme **voisins** l'ensemble

des émetteurs potentiels qui sont définis dans la section 5.4.3.3. Cet ensemble est calculé lors de l'étape de vidage des canaux de communication après partir des tâches qui induisent des communications dans le graphe de flot de données de l'application.

Le paramètre n, le nombre de processus, est connu. Le paramètre v, le nombre de voisins, n'est pas connu. Cette valeur dépend notamment de l'application et de l'ordonnancement utilisé. Dans la suite de cette section, nous cherchons à évaluer la valeur de v pour une application récursive ordonnancée par vol de travail grâce à des mesures.

Nombre de voisins (v)	Proportion de mesures
$v = 0$	14,52 %
$v = 1$	73,83 %
$v = 2$	4,22 %
$v = 3$	2,53 %
$v = 4$	1,60 %
$v = 5$	0,93 %
$v = 6$	0,56 %
$v \geq 7$	1,81 %

Tab. 6.2: Distribution des mesures du nombre de voisins pour une exécution de l'application N-reines sur 1193 machines

Pour effectuer ces mesures, nous utilisons l'application des N-reines que nous exécutons sur 1153 machines de Grid'5000 avec un ordonnancement par vol de travail. À intervalle régulier pendant l'exécution et sur tous les processus participant au calcul, nous comptons le nombre de voisins de chaque processus selon la définition donnée dans la section 5.4.3.3. La distribution des mesures en fonction de la valeur v du nombre de voisins est donnée dans le tableau 6.2. Nous remarquons que plus de 88 % des mesures correspondent à un nombre de voisins de 0 ou 1 ; la valeur maximale relevée est de 37 voisins pour un processus.

Sur la figure 6.7 nous montrons l'évolution du nombre moyen de voisins en fonction du nombre de machines utilisées. Les mesures indiquent un nombre moyen de 1,5 voisin par processus. Cette valeur diminue légèrement lorsque le nombre de machines utilisées augmente.

Dans le cas d'une application récursive ordonnancée par vol de travail, nous pouvons considérer que le nombre moyen de voisins par processus est une constante faible. Ainsi la complexité globale en nombre de messages de notre protocole optimisé de gestion de la cohérence mutuelle apparait comme linéaire fonction du nombre de processus.

6.3.2 Temps de gestion de la cohérence mutuelle

Nous mesurons maintenant le temps de gestion de la cohérence mutuelle sur l'application de N-reines. Comme nous l'avons vu précédemment, le cout de ce protocole dépend essentiellement du nombre de processus participant au calcul.

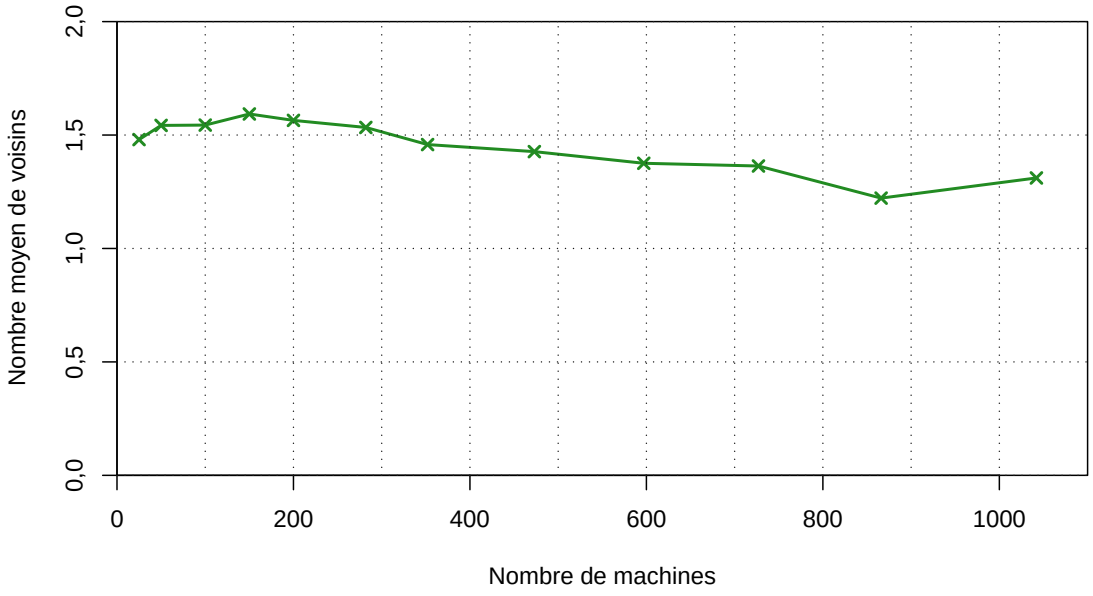


FIG. 6.7: Nombre moyen de voisins durant l'exécution en fonction du nombre de machines utilisées pour l'application des N-reines

Pour réaliser ces mesures nous considérons une reconfiguration vide, c'est-à-dire qui ne fait rien, avec cohérence mutuelle. La fonction de reconfiguration de la reconfiguration vide consiste donc uniquement à l'appel des fonctions de gestion de cohérence mutuelle de KAAPI[2]. Le temps mesuré correspond au temps total d'exécution de la reconfiguration sur le processus maitre, du début du prologue à la fin de l'épilogue.

Les mesures ont été effectuées sur les machines de Grid'5000 en plaçant un processus de calcul par machine. La figure 6.8 montre les temps moyens mesurés pour exécuter la reconfiguration vide en fonction du nombre de machines sur lequel s'exécute l'application. Trois séries de mesures ont été réalisées à trois occasions différentes et en changeant l'ordre d'utilisation des machines ; les résultats de chacune de ces séries de mesures sont donnés dans les sous figures (a), (b) et (c).

La première série de mesures, figure 6.8a a été réalisée sur 1042 machines réparties sur 7 sites de Grid'5000 en utilisant la disposition du tableau ci-dessous.

Machines	1–282	283–352	353–473	474–597	598–727	728–866	867–1042
Site	Orsay	Toulouse	Lille	Sophia	Lyon	Rennes	Bordeaux

Pour les mesures effectuées sur n machines, seules les n premières machines du tableau sont utilisées. Ce tableau indique alors des sites de Grid'5000 utilisés. Par exemple, les mesures sur 473 machines ont utilisé 282 machines du site d'Orsay, 70 machines du site de Toulouse et 121 du site de Lille. Cette information sur la répartition des machines entre les sites est aussi portée sur le bas des figures.

[2] `acquire_mutual_consistency()` et `release_mutual_consistency()`

Pour la deuxième série de mesures, figure 6.8b, nous avons utilisé 1153 machines de Grid'5000 réparties sur 8 sites de la manière suivante.

Machines	1–283	284–351	352–443	444–622	623–737	738–867	868–1008	1009–1153
Site	Orsay	Toulouse	Lille	Bordeaux	Nancy	Lyon	Sophia	Rennes

Pour la troisième série de mesures, figure 6.8c, nous avons utilisé 879 machines de Grid'5000 réparties sur 7 sites, dont la répartition est donnée ci-dessous.

Machines	1–138	139–283	284–403	404–537	538–670	671–748	749–879
Site	Nancy	Sophia	Lille	Bordeaux	Rennes	Toulouse	Lyon

Les courbes de la figure 6.8 montre le temps de gestion de la cohérence mutuelle, version optimisée et version non optimisée, en fonction du nombre de machines et pour différentes conditions expérimentales. Chaque point correspond à la moyenne de 30 à 200 mesures. Les barres d'erreur affichent l'écart type des valeurs mesurées.

Nous nous intéressons tout d'abord aux mesures de la version non optimisée du protocole de gestion de la cohérence mutuelle. Nous constatons une variabilité importante des temps mesurés. En effet, pour un grand nombre de machines (*i.e.* 1000 machines), l'écart type dépasse 0,5 seconde (voire même 1 seconde pour les séries de mesure n° 1 et n° 2).

Dans le cas du protocole non optimisé, le temps de gestion de la cohérence mutuelle augmente avec le nombre de machines. À cause de la variabilité des mesures, il est difficile de préciser le comportement des courbes. Cependant, il semble que ce temps évolue par paliers. L'ajout des machines d'un site supplémentaire semble provoquer une augmentation brutale du temps de gestion de la cohérence mutuelle. Ce phénomène est assez visible avec le site de Bordeaux dans les trois séries de mesures. L'ajout de machines proches (géographiquement, mais surtout du point de vue de la qualité du réseau) a donc un impact beaucoup plus faible que l'ajout de machines distantes.

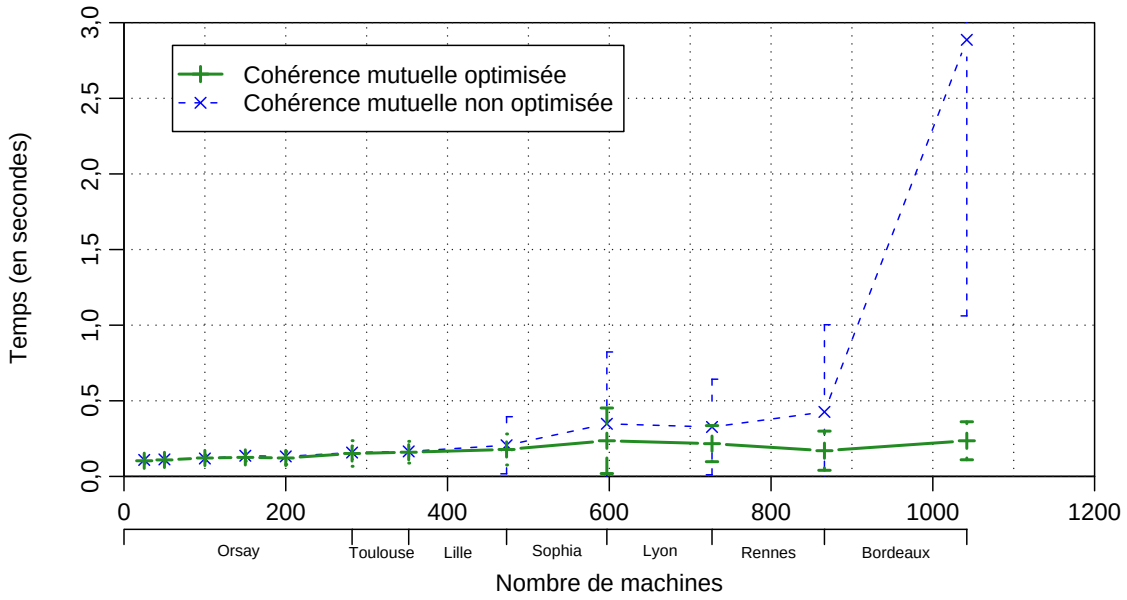
Pour la version optimisée de ce protocole, les valeurs mesurées sont beaucoup plus stables. Les écarts types sont la plupart du temps inférieurs à 0,1 seconde[3].

De plus, l'influence du nombre de machines sur le temps de gestion de la cohérence mutuelle est très faible. Les courbes sont quasiment plates. Pour la première série de mesures, le temps passe de 0,10 s pour 25 machines à 0,24 s pour 1042 machines ; pour la deuxième série de mesures, il passe de 0,12 s pour 100 machines à 0,19 s pour 1153 machines ; et pour la troisième série de mesures, il passe de 0,76 s pour 69 machines à 0,97 s pour 879 machines.

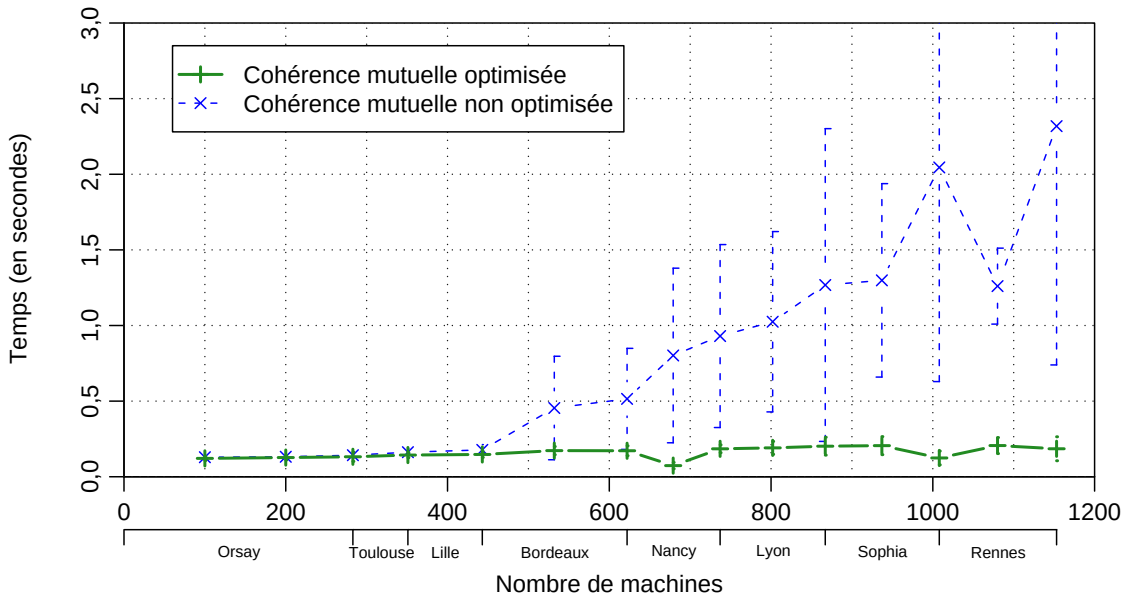
Pour la troisième série de mesures, le temps de cohérence mutuelle est, dès les premières machines, beaucoup plus élevé que pour les deux autres séries de mesures. Nous n'avons pas trouvé d'explication à ce phénomène autre que les différences de conditions expérimentales. Cependant, nous retrouvons globalement les mêmes comportements que pour les autres séries de mesures.

Pour résumer, comparée à la version non optimisée, la version optimisée du protocole offre de bonnes performances. Le temps de gestion de la cohérence mutuelle est stable

[3]Sur les figures, l'écart type n'est pas tracé lorsqu'il est trop faible.

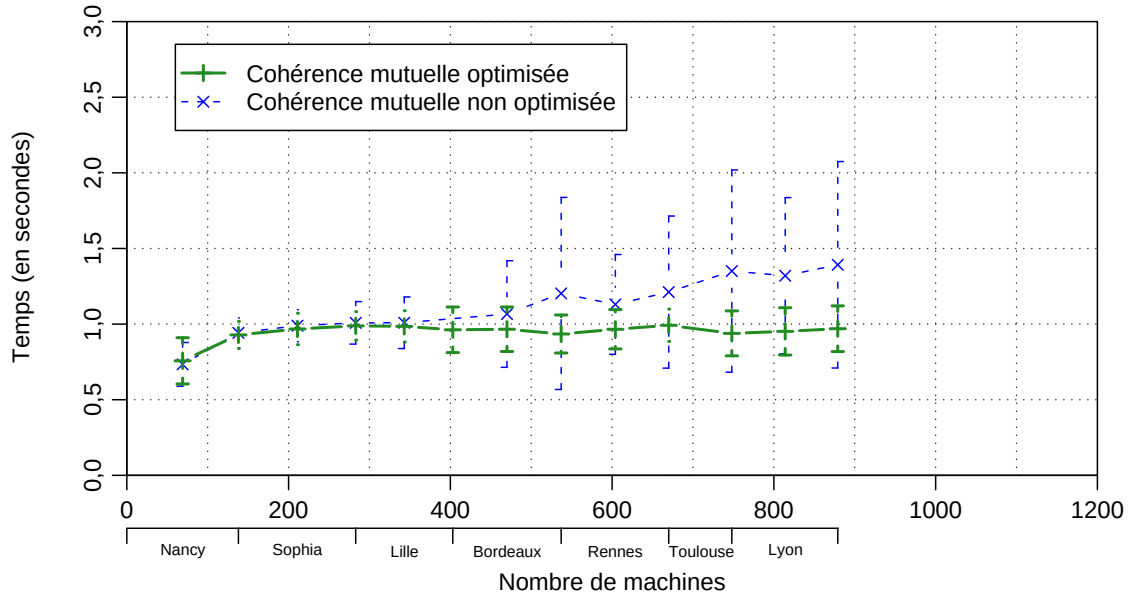


(a) Série de mesures n° 1



(b) Série de mesures n° 2

FIG. 6.8: Temps de gestion de la cohérence mutuelle en fonction du nombre de machines, version optimisée et version non optimisée



(c) Série de mesures n° 3

FIG. 6.8: Temps de gestion de la cohérence mutuelle en fonction du nombre de machines, version optimisée et version non optimisée

et l'ajout de machines supplémentaires, même à partir de sites distants, n'a qu'une influence minime. Il permet ainsi de coordonner un ensemble de 1000 machines en moins de 250 ms. Tous ces éléments nous poussent à croire que ce protocole optimisé se comportera bien sur un nombre plus important de machines.

6.4 Conclusion

Dans ce chapitre, nous avons présenté plusieurs séries d'expériences qui nous ont permis d'évaluer les mécanismes de reconfiguration que nous avons proposés au chapitre 5. Ces expériences ont porté sur deux aspects : la gestion des accès concurrents et la gestion de la cohérence mutuelle.

Les expériences sur la gestion des accès concurrents se sont tout d'abord attachées à comparer les performances de l'exécution concurrente et de l'exécution coopérative pour la reconfiguration « vol de travail » dans le moteur d'exécution X-KAAPI. La version coopérative du vol de travail se révèle plus efficace à grain fin puisqu'elle évite d'ajouter des instructions couteuses de synchronisation sur le chemin critique de l'application. Cependant, à gros grain, l'exécution coopérative est victime d'une latence importante pour répondre aux requêtes de vol. En conséquence, les processeurs inactifs mettent plus de temps pour récupérer du travail.

Nous avons ensuite comparé les performances du vol de travail coopératif de X-KAAPI avec le vol de travail de CILK et TBB sur des algorithmes de la STL. Les gains

mesurés sur 8 cœurs sont assez importants, même sur des tableaux de petite taille. L'exécution coopérative semble être une technique idéale pour l'exécution à grain fin avec un faible surcout.

La suite de ce chapitre a évalué le cout de la gestion de la cohérence mutuelle à grande échelle. Tout d'abord, nous avons étudié le nombre moyen de voisins par processus pour une application ordonnancée par vol de travail. Ce nombre est en moyenne faible (inférieur à 2) et permet de justifier la version optimisée du protocole de gestion de cohérence mutuelle qui repose sur le modèle de graphe de flot de données de Kaapi. Cette version optimisée du protocole de gestion de la cohérence mutuelle permet de coordonner plus de 1000 processus en un temps inférieur à 250 ms. Un autre avantage est que le temps de coordination est beaucoup plus stable qu'avec le protocole non optimisé.

III

Tolérance aux fautes pour un modèle graphe de flot de données

Tolérance aux fautes dans Kaapi 7

Sommaire

7.1 Introduction

Ce chapitre présente un aperçu général des mécanismes de tolérance aux fautes intégrés dans Kaapi. Il est constitué de deux sections principales.

La section 7.2 introduit les cinq composants grâce auxquels le moteur d'exécution Kaapi peut tolérer les fautes et elle présente leur organisation et leurs interactions. Ces composants sont le détecteur de panne, le coordinateur des pannes, la mémoire stable, le protocole de sauvegarde et le protocole de reprise.

La section 7.3 détaille le composant de sauvegarde coordonnée sur lequel repose les techniques de reprise globale et de reprise partielle qui sont présentées dans les chapitre 8 et 9. Des expérimentations sont également proposées : leurs résultats permettent de mettre en évidence les différentes variables qui influencent le cout de ce mécanisme de sauvegarde coordonnée.

7.2 Organisation de la tolérance aux fautes dans Kaapi

Nous distinguons deux modes de fonctionnement du moteur d'exécution KAAPI vis-à-vis de la tolérance aux fautes : l'exécution sans panne et la reprise.

L'**exécution sans panne** est le mode de fonctionnement durant lequel l'application s'exécute normalement tant qu'aucune panne ne se produit. Durant l'exécution sans panne, il est nécessaire de sauvegarder certaines informations pour assurer la tolérance aux fautes. Cette sauvegarde d'informations peut prendre la forme de sauvegarde de l'état des processus ou de journalisation des messages[1]. Elle doit contenir les informations nécessaires pour effectuer une reprise et utilise une mémoire stable pour conserver les informations sauvegardées.

La **reprise** est le mode de fonctionnement qui intervient après qu'une panne s'est produite. Dans ce mode de fonctionnement, il n'y a pas de sauvegarde d'informations, mais le mécanisme de reprise est activé pour rétablir un état global cohérent de l'application après la panne. Ce protocole utilise les informations sauvegardées sur la mémoire stable pour reconstruire l'état de l'application.

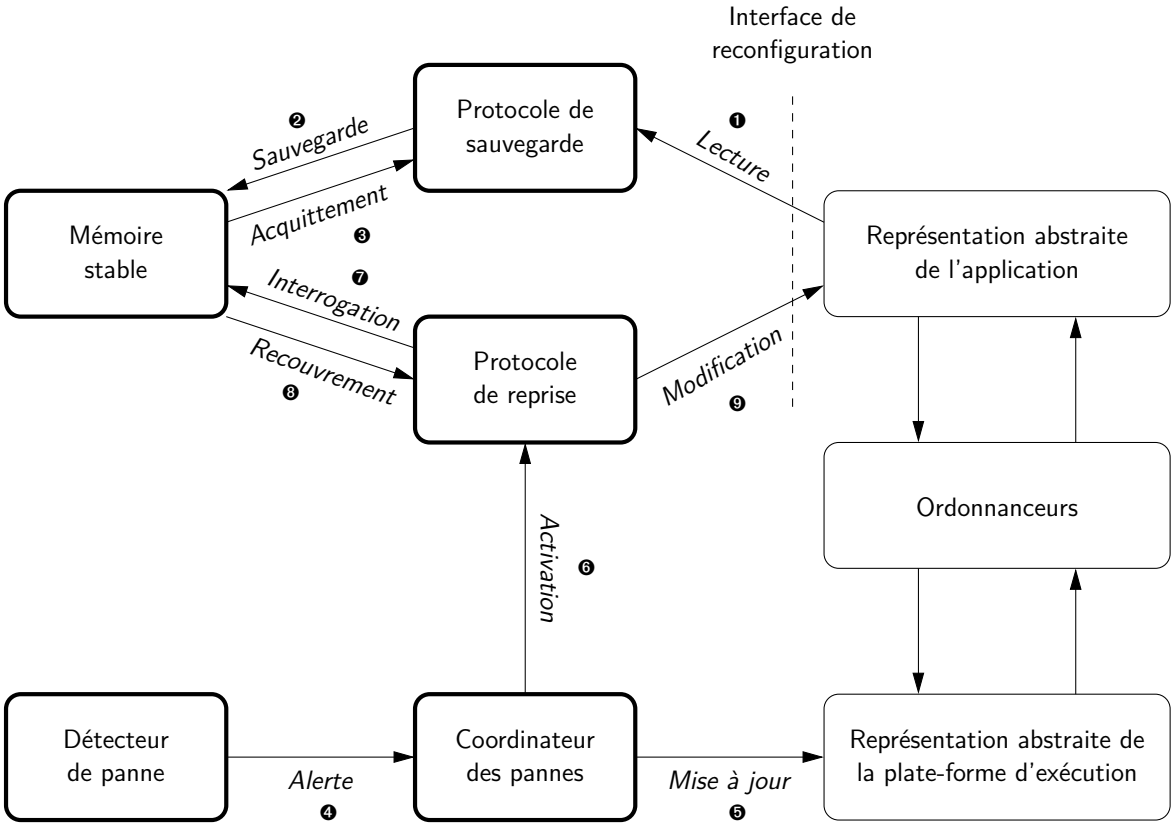


FIG. 7.1: Organisation des mécanismes de tolérance aux fautes dans KAAPI. La signification des numéros est détaillée dans le texte.

[1]Dans la suite de ce chapitre, nous désignerons indifféremment la sauvegarde de l'état des processus ou la journalisation des messages par les termes « sauvegarde d'informations » ou « sauvegarde ».

La tolérance aux fautes dans KAAPI repose sur cinq composants : la **mémoire stable**, le **protocole de sauvegarde**, le **protocole de reprise**, le **détecteur de panne** et le **coordinateur des pannes**. La figure 7.1 présente l'organisation de ces différents composants. Ce schéma est une extension de la figure 4.1 page 72 du chapitre présentant le modèle de programmation ATHAPASCAN et le moteur d'exécution KAAPI.

Durant l'exécution sans panne, le protocole de sauvegarde décide régulièrement de sauvegarder certaines informations. En reprenant la notation des numéros de la figure 7.1, nous avons les opérations suivantes qui sont effectuées.

❶ Tout d'abord, le protocole de sauvegarde lit l'état de l'application qui est contenu dans la représentation abstraite de l'application.

❷ Puis, il sauvegarde ces informations dans la mémoire stable.

❸ Le protocole de sauvegarde reçoit des acquittements de la part du composant de mémoire stable qui lui indiquent que la sauvegarde a bien été reçue et enregistrée.

Lorsqu'une panne se produit, l'exécution passe en mode reprise.

❹ Le détecteur de panne alerte le coordinateur de pannes.

❺ Le coordinateur de pannes met à jour la représentation abstraite de la plate-forme d'exécution pour indiquer les processus défaillants.

❻ Puis il active le protocole de reprise.

❼ Le protocole de reprise interroge la mémoire stable pour connaitre les états sauvegardés et choisir les informations à récupérer.

❽ La mémoire stable envoie les informations demandées.

❾ Le protocole de reprise peut alors reconstruire un état global cohérent de l'application.

Une fois que le protocole de reprise s'est terminé, l'exécution sans panne peut reprendre.

Chacun de ces composants réalise une fonction élémentaire nécessaire pour assurer la tolérance aux fautes. La suite de ce chapitre décrit ces composants, le service rendu ainsi que les hypothèses nécessaires à leur bon fonctionnement.

7.2.1 Détection des pannes

Le but d'un détecteur de panne est de détecter les pannes qui peuvent survenir durant l'exécution de l'application aussitôt que possible. Une fois qu'une panne a été détectée, le détecteur de panne avertit le coordinateur des pannes. Les travaux de cette thèse ne portent pas sur les détecteurs de pannes. Cependant, il est indispensable de s'attaquer à ce problème pour offrir un service de tolérance aux fautes. Dans cette section, nous indiquons uniquement les solutions qui ont été mises en œuvre dans le moteur d'exécution KAAPI.

Dans notre modèle, nous prenons en compte les pannes en termes de processus. Nous considérons qu'un processus est en panne lorsqu'il ne fournit plus le service qu'on attend de lui. Le rôle du détecteur de panne est alors de vérifier que les processus remplissent bien leur rôle.

Cette tâche est assez compliquée en pratique et pour détecter les pannes, nous préférons utiliser plusieurs détecteurs de pannes qui vont chacun tester une propriété des processus. Si la propriété vérifiée par l'un des détecteurs de panne n'est pas vraie pour

un processus donné, alors ce processus est considéré comme défaillant. Le coordinateur des pannes en est alors informé.

Nous sommes conscient que l'approche choisie est pragmatique. Ce choix s'explique d'une part par la difficulté de récupérer un code de détecteur (en particulier du code C++, plus simple à intégrer dans Kaapi entièrement en C++) et d'autre part par le manque de temps par rapport à nos objectifs de ce travail. Idéalement, ce composant devrait passer à l'échelle [25] et offrir un certain niveau de qualité de service [53].

Dans le moteur d'exécution Kaapi, nous avons implémenté deux détecteurs de pannes : un détecteur d'erreur réseau et un détecteur par battement de cœur.

- Le **détecteur d'erreur réseau** est un détecteur de panne qui s'exécute sur tous les processus de calcul Kaapi. Ce détecteur intercepte tous les codes d'erreurs en provenance des différentes couches de communication. Lorsqu'il intercepte une erreur inattendue (échec de l'émission d'un message, message reçu incomplet, fermeture d'une connexion, etc.), le processus qui est à l'autre bout du canal de communication est considéré comme défaillant.
  Ce détecteur vérifie donc la propriété qui dit qu'aucune communication ne doit produire de code d'erreur inattendue. Cela nécessite donc d'identifier tous les codes d'erreur et leurs conséquences pour le moteur d'exécution. Il permet notamment de détecter les processus qui s'arrêtent brutalement (car dans ce cas le système d'exploitation ferme la connexion restée ouverte). Ce détecteur est intéressant en pratique puisqu'il offre une bonne réactivité.
- Le **détecteur par battement de cœur**, également appelé *heartbeat*, s'exécute sur tous les processus de calcul. Le principe de ce détecteur est d'émettre périodiquement un message HB-PING vers un autre processus choisi au hasard. À la réception d'un tel message, le processus distant doit répondre avec le HB-PONG. Si le message de réponse HB-PONG n'est pas reçu dans un délai configurable, alors le processus distant est considéré comme défaillant.
  Ce détecteur vérifie donc la propriété qui dit que tous les processus doivent être joignables et répondre dans le délai imparti. Il permet de détecter des pannes qui peuvent ne pas être repérées par le détecteur d'erreur réseau, par exemple le cas de la défaillance du système d'exploitation ou la coupure physique d'un lien réseau.
  Ce détecteur peut cependant produire des faux positifs dans le cas où le réseau ou la machine distante sont surchargées. C'est notamment le cas si le délai de réponse est trop faible. Inversement, un délai trop important entrainera une perte de réactivité en cas de panne, ce qui a pour conséquence d'augmenter le temps de calcul gaspillé.

7.2.2 Réaction aux pannes

La réaction aux pannes est réalisée par le composant coordinateur des pannes. Ce composant assure les trois fonctions suivantes.

- Il collecte les alertes de pannes en provenance des détecteurs de pannes.
- Il informe tous les processus de calcul de la défaillance pour qu'ils puissent mettre à jour leur représentation abstraite de la plate-forme d'exécution. Ceci permet

ensuite d'ignorer toute communication (entrante ou sortante) avec un processus considéré défaillant.
- Enfin, il active le protocole de reprise.

Le coordinateur des pannes ne s'interroge pas sur la validité des pannes détectées, mais il doit mettre en place les actions nécessaires à la reprise de l'exécution.

Il faut remarquer que dans notre modèle, nous considérons que les pannes sont franches. Ainsi, même si la panne est seulement temporaire ou si un détecteur de panne commet une erreur, nous choisissons tout de même d'exclure définitivement le processus de la suite du calcul. C'est pour cela que tous les autres processus doivent être informés des défaillances avant la reprise du calcul. Ainsi à la reprise, tous les processus non défaillants ignoreront le ou les processus considérés comme défaillants pour éviter qu'ils ne corrompent l'état de l'application. Ces considérations permettent aussi de traiter le cas des pannes byzantines.

En général, un processus ne tombe pas toujours en panne tout seul. En effet, les défaillances de processus peuvent être groupées, par exemple si toute une grappe de machines est coupée du réseau ou si un premier processus défaillant émet des données qui provoquent des erreurs sur d'autres processus.

C'est pourquoi le coordinateur de pannes implémenté dans le moteur d'exécution KAAPI utilise un délai de garde, qui permet d'attendre d'autres pannes avant de déclencher la reprise. Ce délai permet d'attendre un certain temps avant d'activer le protocole de reprise, de manière à éviter que les autres pannes ne soient détectées pendant la reprise.

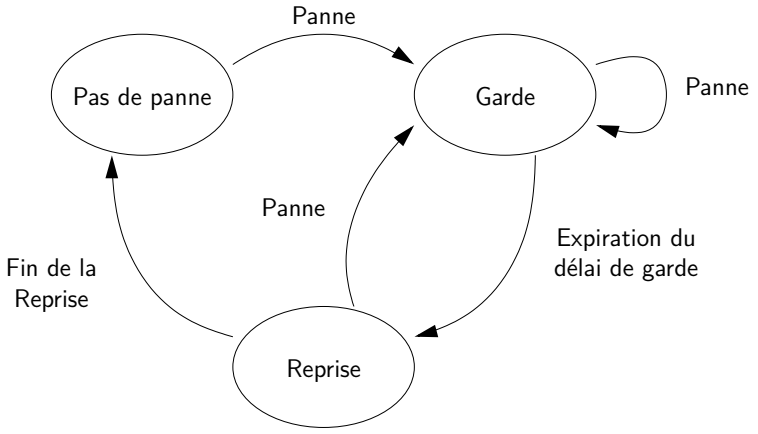


FIG. 7.2: Réaction aux pannes dans KAAPI

La figure 7.2 montre le diagramme de décision du coordinateur des pannes dans KAAPI. Le coordinateur des pannes peut se trouver dans trois états différents : Pas de panne, Garde et Reprise.

- Lorsqu'aucune panne ne s'est produite, l'état est Pas de panne. Dans cet état, soit l'application est en cours d'exécution, ou bien elle est en train de sauvegarder son état.
- L'apparition d'une panne fait passer le coordinateur des pannes dans l'état Garde : il est en attente du délai de garde. De plus, chaque alerte de panne réinitialise le

délai de garde et provoque la diffusion de l'information de la nouvelle panne à tous les processus de calcul.
- La fin du délai de garde provoque le passage dans l'état Reprise. Le passage dans cet état correspond à l'activation du protocole de reprise. Lorsque la reprise est terminée, l'exécution de l'application reprend et l'état redevient Pas de panne.

Il faut noter que le coordinateur des pannes ne réagit qu'après le délai de garde suite au signalement de la première panne d'un processus. Si un processus en panne est de nouveau signalé comme étant défaillant, cette alerte est ignorée.

7.2.3 Mémoire stable

La mémoire stable est une abstraction qui permet de désigner un moyen sûr de conserver les informations nécessaires pour redémarrer l'application en cas de panne. La mémoire stable est le seul composant qui, en théorie, doit ne pas tomber en panne. En effet, c'est ce composant qui réalise le transfert fiable d'informations entre l'exécution sans panne et la reprise.

Nous identifions quatre opérations élémentaires que le composant de mémoire stable doit fournir pour réaliser correctement sa fonction.
- La sauvegarde : le protocole de sauvegarde enregistre des informations dans la mémoire. À chaque information est associé un identifiant qui est donné par le protocole de sauvegarde.
- L'acquittement : la mémoire stable peut fournir pour chaque information sauvegardée un accusé de réception qui confirme que l'information a bien été reçue et qu'elle est maintenant conservée de manière stable.
- L'interrogation : le protocole de reprise interroge la mémoire stable pour connaitre les informations conservées en mémoire stable. Cela consiste à consulter la liste des identifiants des informations sauvegardées sans nécessairement récupérer les informations elles-mêmes.
- Le recouvrement : le protocole de reprise choisit certaines informations et les récupère à partir de la mémoire stable.

La principale difficulté pour réaliser une mémoire stable est de garantir la stabilité des informations. La stabilité en-soi n'est pas réalisable, mais il est possible de s'en approcher à l'aide de diverses techniques. Il faut prendre en compte plusieurs aspects comme le stockage physique des données (disque dur, mémoire vive, etc.) ou leur accessibilité (accès réseau). Ainsi, les principales techniques sont le codage redondant et la réplication des données [121]. Elles permettent d'éviter de reposer sur un unique composant centralisé plus sensible aux pannes. Cependant, ces considérations sont en dehors du cadre de cette thèse. La plupart du temps, la mémoire stable est uniquement une simple hypothèse sur la machine et le processus qui offre ce service.

Outre ces aspects, la performance est certainement le point le plus important pour un composant de mémoire stable. En effet, la principale fonction d'une mémoire stable est de recevoir et d'émettre des données. Lorsque le volume de données à sauvegarder et à restituer est important (ce qui est par exemple le cas des applications de calcul scientifique), la performance de la mémoire stable est l'élément qui va principalement influencer les durées de sauvegarde et de reprise [43]. Ces paramètres ont alors un impact sur le nombre de sauvegardes ainsi que sur le travail perdu en cas de panne [41].

De plus, le placement physique de ce composant sur la plate-forme d'exécution a une importance capitale.

Dans le moteur d'exécution KAAPI, la mémoire stable est réalisée par un ensemble de processus spéciaux appelés serveurs de sauvegarde. Chaque processus de calcul est associé à un serveur de sauvegarde et chaque serveur de sauvegarde peut être utilisé par plusieurs processus de calcul. Les serveurs de sauvegarde conservent les informations que le processus de calcul lui envoie par le biais du protocole de sauvegarde.

Ces serveurs de sauvegarde utilisent l'espace disque de la machine sur laquelle ils s'exécutent pour conserver les données et les méta-données[2]. En cas de défaillance du processus serveur de sauvegarde, il est donc possible de redémarrer le processus sans perdre les informations sauvegardées.

Cependant, les serveurs de sauvegarde KAAPI n'utilisent pas de mécanismes de redondance des données. Donc leur stabilité repose essentiellement sur l'hypothèse de stabilité du matériel et de la connexion utilisée, et il sera important dans l'avenir de supprimer cette hypothèse. Actuellement, les serveurs de sauvegarde KAAPI s'apparentent plus à une « simulation » de mémoire stable destinée à réaliser les expériences.

7.2.4 Protocoles de sauvegarde et de reprise

Le protocole de sauvegarde est le composant qui est responsable de sélectionner et d'enregistrer les informations nécessaires au redémarrage. Ce composant est actif durant l'exécution sans panne de l'application. Le composant du protocole de reprise est quant à lui est chargé de redémarrer l'application en utilisant les informations sauvegardées sur la mémoire stable par le protocole de sauvegarde. Ce composant est activé lors de la phase de reprise.

De nombreuses techniques existent pour réaliser la sauvegarde et la reprise d'une application. Ces techniques ont été présentées dans l'état de l'art à la section 2.4. Généralement, la sauvegarde et la reprise sont fortement couplées dans la mesure où la méthode pour redémarrer l'application dépend fortement des informations qui ont été sauvegardées durant l'exécution. Cependant, nous décidons de distinguer un composant de sauvegarde et un composant de reprise parce qu'ils effectuent chacun des opérations différentes et qu'ils s'exécutent à des moments différents.

Ces deux composants sont les seuls qui vont directement interagir avec l'état de l'application et son exécution. Nous proposons d'effectuer ces interactions (lecture ou écriture), en utilisant les mécanismes de reconfiguration décrits dans la partie II de cette thèse. Ceci permet simplifier l'implémentation de ces protocoles et de bénéficier d'un accès sûr à l'état de l'application.

Dans le moteur d'exécution KAAPI, deux familles de protocoles de sauvegarde et de reprise sont actuellement implémentés.

- La famille de protocoles **TIC** (*Theft-Induced Checkpointing*) comporte deux composants : un composant de sauvegarde appelé TIC-Checkpoint, et un composant de reprise appelé TIC-Restart. Ces deux composants s'utilisent ensemble pour

[2]Les méta-données correspondent, entre autres, aux identifiants et aux fichiers physiques des informations sauvegardées.

rendre le moteur d'exécution KAAPI tolérant aux fautes pour les applications utilisant l'ordonnancement par vol de travail. Ils implémentent le protocole TIC proposé par Jafar dans [92] qui est une adaptation pour le vol de travail de la technique de sauvegarde induite par les communications (section 2.4.2.3).

- La famille de protocoles **CCK** (*Coordinated Checkpointing in* KAAPI) comporte un composant de sauvegarde appelé CCK-Checkpoint et deux composants de reprise CCK-GlobalRestart et CCK-Restart.
  Le composant de sauvegarde CCK-Checkpoint est une implémentation de la technique classique de sauvegarde coordonnée bloquante présentée à la section 2.4.2.2. Ce composant est implémenté en utilisant le mécanisme de reconfiguration dynamique présenté dans la partie II en imposant une contrainte de cohérence mutuelle (section 5.4.3) sur l'ensemble des processus qui participent au calcul.
  Les deux composants de reprise sont capables de redémarrer une application sauvegardée par CCK-Checkpoint. Le composant CCK-GlobalRestart effectue la reprise avec la technique classique où tous les processus redémarrent depuis la dernière sauvegarde. Il est présenté et illustré par des expérimentations au chapitre 8. Le composant CCK-Restart offre un redémarrage optimisé qui permet seulement à une partie de processus de repartir depuis leur dernière sauvegarde. Ce dernier protocole est détaillé dans le chapitre 9.

Maintenant que nous avons présenté l'ensemble des composants nécessaires aux mécanismes de tolérance aux fautes dans KAAPI, nous allons présenter en détail la réalisation de la sauvegarde coordonnée 2.4.2.2 sur laquelle se basent les protocoles de reprise CCK.

7.3 Sauvegarde coordonnée

Pour réaliser la sauvegarde coordonnée, nous choisissons d'utiliser le mécanisme de reconfiguration dynamique présenté au chapitre 5. Ce mécanisme de reconfiguration permet d'accéder de manière sure à l'état de chacun des processus. De plus, le mécanisme de cohérence mutuelle de la section 5.4.3 permet de garantir la cohérence et l'accessibilité de l'ensemble des états locaux des processus qui composent l'application.

Le protocole de sauvegarde CCK repose donc sur la synchronisation offerte par le mécanisme de cohérence mutuelle. Ceci permet de simplifier l'implémentation de la sauvegarde. Ainsi le protocole de sauvegarde résultant s'apparente à la technique de sauvegarde coordonnée bloquante car l'état des processus ne sera sauvegardé qu'une fois que tous les canaux de communication auront été vidés.

7.3.1 État de l'application et état local d'un processus

L'état d'une application est constitué de l'état de tous ses processus et de l'état des canaux de communication [49]. De plus, le mécanisme de cohérence mutuelle utilisé pour la sauvegarde permet de rendre l'état accessible, c'est-à-dire de vider les canaux de communication. L'état de l'application est alors composé uniquement de l'état local de chacun des processus. Pour réaliser une sauvegarde de l'état de l'application, il convient

donc de sauvegarder l'état de tous les processus composant l'application après les avoir coordonnés pour les rendre mutuellement cohérents entre eux.

L'état d'un processus peut être sauvegardé grâce à sa représentation abstraite sous la forme d'un graphe de flot de données et de ses attributs. Ce graphe comprend notamment les tâches et les versions des données d'entrée. Au moment de la sauvegarde (*i.e.* au point local de reconfiguration sur chaque processus), l'exécution est arrêtée et aucune tâche n'est en cours d'exécution ; donc comme nous l'avons expliqué dans la section 4.3.1, le graphe de flot de données et ses attributs constituent donc une représentation valide de l'état de l'application.

L'état global de l'application correspond donc à l'ensemble des graphes de flot de données locaux. Un point de sauvegarde de l'application est donc constitué d'une copie de l'ensemble des graphes de flot de données locaux des processus qui participent au calcul à l'instant de la sauvegarde.

De plus, la sauvegarde sous forme d'un graphe de flot de données plutôt que sous forme de zone mémoire d'un processus et de ses registres permet de résoudre (en partie) le problème de l'hétérogénéité. Un tel point de sauvegarde pourra être rechargé sur n'importe quelle machine, quelle que soit son architecture, pour peu que les données soient bien encodées.

7.3.2 Protocole de sauvegarde

Pour effectuer une étape de sauvegarde, nous distinguons deux fonctions élémentaires : la fonction de **coordination** et la fonction de **sauvegarde locale**. La fonction de coordination est réalisée par un unique processus appelé processus coordinateur tandis que la fonction de sauvegarde locale est réalisée par tous les processus qui participent au calcul.

En pratique, le processus coordinateur peut être un des processus de calcul, mais il a la particularité d'initier l'étape de sauvegarde. Dans Kaapi et pour des raisons de simplicité, ce processus est systématiquement le processus qui possède l'identifiant 0.

7.3.2.1 Protocole sur le processus coordinateur

Le rôle du processus coordinateur de la sauvegarde est de coordonner tous les processus pour réaliser une étape de sauvegarde et de vérifier le bon déroulement de la sauvegarde. Nous réalisons cette sauvegarde en utilisant le mécanisme de reconfiguration dynamique présenté au chapitre 5. Nous considérons ici la sauvegarde comme une reconfiguration qui ajoute la propriété de tolérance aux fautes à l'application. Plus précisément, la reconfiguration « sauvegarde » reconfigure l'application en une application qui ne perdra pas plus de $W_t - W_{sauvegarde}$ en cas de panne, où W_t est le travail effectué à l'instant t et $W_{sauvegarde}$ est le travail effectué au moment de la sauvegarde.

Le protocole de sauvegarde sur le processus coordinateur est défini à travers les trois étapes qui constituent une reconfiguration : le prologue, la réalisation et l'épilogue. Pour l'exécuter, le processus coordinateur utilise un processus léger dédié pour ne pas bloquer le calcul pendant le prologue et l'épilogue. Le protocole de sauvegarde est déclenché à des dates prédéfinies (généralement de manière périodique) mais peut éventuellement

être forcé par un appel direct, par exemple en cas d'arrêt forcé de l'application par l'utilisateur.

Prologue. Durant le prologue, le processus coordinateur choisit la version de la sauvegarde et définit la reconfiguration « sauvegarde » qui sera appliquée sur les processus de calcul. La **version** de la sauvegarde est un identifiant unique qui doit permettre de reconnaitre une sauvegarde donnée. La reconfiguration est définie par les éléments suivants.

- La fonction de reconfiguration « sauvegarde locale » qui est la fonction qui sera exécutée sur chacun des processus de calcul pour sauvegarder l'état local de l'application.
- La cible de la reconfiguration qui dans le cas de la sauvegarde coordonnée de l'application correspond à l'ensemble de tous les processus qui participent au calcul.
- Le seul paramètre d'entrée de la fonction de reconfiguration est la version de la sauvegarde.

Le prologue aboutit alors à la réalisation de la sauvegarde.

Réalisation. La réalisation correspond à l'exécution de la fonction de reconfiguration « sauvegarde locale » sur tous les processus de calcul. Cette fonction est détaillée dans la section suivante 7.3.2.2. La réalisation se termine lorsque tous les processus de calcul ont terminé la sauvegarde locale.

Épilogue. L'épilogue consiste à définir le statut de la sauvegarde effectuée. L'épilogue permet de vérifier que l'exécution de la sauvegarde locale s'est exécutée correctement sur chacun des processus. Ensuite l'épilogue attend la réception d'un acquittement en provenance de la mémoire stable pour chacun des processus ayant participé à la sauvegarde. L'acquittement signifie que la sauvegarde d'un processus a bien été reçue et assure que les informations sauvegardées sont bien conservées sur la mémoire stable.

Si tout s'est déroulé correctement la sauvegarde peut être validée. L'information de validité d'une sauvegarde est également inscrite dans la mémoire pour permettre de reconnaitre la dernière sauvegarde valide lorsqu'un redémarrage est nécessaire. Les anciennes sauvegardes peuvent éventuellement être supprimées de la mémoire stable puisqu'avec un protocole de sauvegarde coordonnée, il est toujours possible de redémarrer à partir de la dernière sauvegarde.

Cette étape d'attente ne ralentit pas le calcul puisque le protocole de sauvegarde est exécuté dans un processus léger dédié. Elle permet de simplifier la gestion des différentes versions des sauvegardes : il n'est plus nécessaire d'avoir un ramasse-miette distribué pour supprimer les sauvegardes inutiles et, la recherche de la dernière sauvegarde valide est immédiate.

7.3.2.2 Protocole sur les processus de calcul

Sur les processus de calcul, le protocole de sauvegarde consiste à exécuter la fonction de reconfiguration « sauvegarde locale » définie par le processus coordinateur. Cette fonction effectue les opérations suivantes.

1. Elle explore l'état de l'application en utilisant l'interface de gestion des accès concurrents dont les propriétés ont été décrites à la section 5.3 du chapitre 5. Tout d'abord la reconfiguration « sauvegarde locale » prend connaissance de tous les K-threads existants puis elle les marque (action MARK) pour les désigner comme des objets visés par la reconfiguration.
2. Elle fait à la fonction `acquire_mutual_consistency()` pour obtenir la cohérence mutuelle du processus vis-à-vis de tous les autres. Ceci provoque entre autres l'arrêt des flots d'exécution.
3. Après l'appel à `acquire_mutual_consistency()`, le processus est dans son point local de reconfiguration. L'état de l'application est donc sauvegardé en envoyant une copie de chaque K-thread sur la mémoire stable sous le numéro de version donné pour cette sauvegarde. La sauvegarde se termine par une demande d'émission d'un acquittement vers le processus coordinateur.
4. L'appel à `release_mutual_consistency()` permet de terminer le point local de reconfiguration.
5. Les objets marqués précédemment sont démarqués (action UNMARK).
6. Enfin, la fonction de reconfiguration « sauvegarde locale » définit le résultat de l'exécution de la sauvegarde qui sera retourné au processus coordinateur pour indiquer si la sauvegarde s'est déroulée correctement. Ce résultat est utilisé par le processus coordinateur au moment de l'épilogue pour vérifier que le protocole de sauvegarde s'est bien déroulé correctement.

7.3.3 Expérimentations

Cette section présente quelques expériences qui visent à démontrer l'effet des certains paramètres sur la durée d'une sauvegarde coordonnée et sur le temps d'exécution d'une application utilisant la tolérance aux fautes. Nous mettons en avant l'influence que chacun de ces paramètres peut avoir sur le cout d'une sauvegarde.

Ces expériences sont réalisées à l'aide du moteur d'exécution Kaapi présenté au chapitre 4.

7.3.3.1 Influence des serveurs de sauvegarde

Sur les grappes ou les grilles de calcul qui ne possèdent pas de machine de stockage dédiée, il est possible d'utiliser les machines de calcul pour réaliser les serveurs de sauvegarde. Ces serveurs de sauvegarde servent à conserver les informations sauvegardées par plusieurs processus de calcul.

Pour obtenir de bonnes performances lors de l'étape de sauvegarde, il est important de choisir un placement et un nombre de serveurs adéquat. Ce placement dépend fortement de la topologie du réseau de la grappe ou de la grille de calcul. Nous proposons une expérience qui met en évidence ce phénomène.

Cette expérience a été réalisée sur un sous-ensemble de 180 machines de la grappe d'Orsay de Grid'5000. L'architecture du réseau de cette grappe est montrée à la figure 7.3. La topologie réseau de l'ensemble des nœuds considérés est organisée en deux niveaux de *switch*s. Chacun des 12 *switch*s du premier niveau connecte entre elles 15 machines

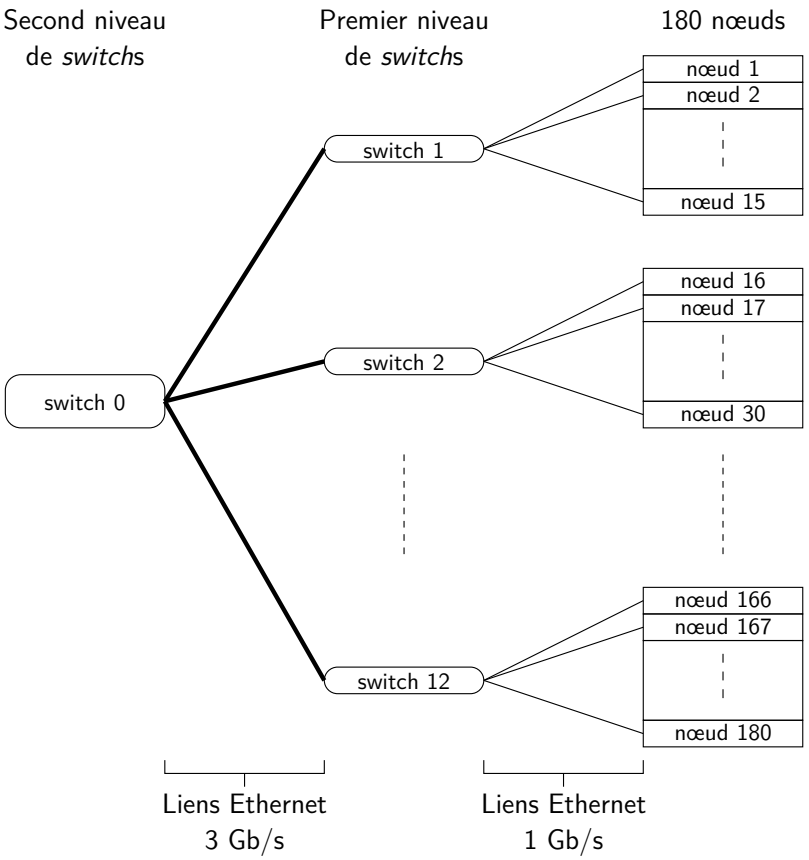


FIG. 7.3: Architecture du réseau d'un sous-ensemble de nœuds d'Orsay de Grid'5000

par des liens Ethernet de 1 Gb/s ; Le *switch* du second niveau connecte entre eux les *switch*s du premier niveau par des liens de 3 Gb/s.

Parmi les 180 machines utilisées, 120 sont utilisées pour exécuter des processus de calcul. Les autres sont utilisées pour exécuter un nombre variable (12, 24 ou 60) de serveurs de sauvegarde. À chaque processus de calcul est associé un serveur de sauvegarde sur lequel il enregistrera son état lors de l'étape de sauvegarde. Un serveur de sauvegarde est donc partagé par 10, 5 ou 2 processus de calcul selon le nombre de serveurs de sauvegarde choisi. Nous comparons alors trois méthodes de placement des serveurs de sauvegarde parmi l'ensemble des machines de la grappe.

- Le **placement dans l'ordre** est l'approche naïve qui consiste à placer les serveurs selon l'ordre par défaut donné par le gestionnaire de ressources. Ainsi, les premières machines constituent les machines de calcul et les dernières machines constituent les serveurs de sauvegarde. Avec un tel placement, tous les serveurs de sauvegarde sont regroupés sur quelques *switch*s de premier niveau.
- Le **placement par *switch*** correspond à répartir les serveurs de sauvegarde équitablement parmi tous les *switch*s disponibles. De plus, nous prenons soin d'associer à chaque à chaque processus de calcul un serveur de sauvegarde du même *switch*. Cette approche nécessite cependant d'avoir la connaissance de l'organisation du réseau au sein de la grappe.
- Le **placement aléatoire** consiste à définir aléatoirement le rôle de chaque machine de la grappe. Avec une grande probabilité, les serveurs de sauvegardes sont répartis de manière équilibrée entre les *switch*s, cependant le serveur de sauvegarde associé à chaque processus de calcul est potentiellement placé sur un autre *switch*.

Pour réaliser cette expérience, nous utilisons une application de type décomposition de domaine. À chaque étape de sauvegarde, le volume total (*i.e.* pour tous les processus) des données à sauvegarder représente environ 20 Go, soit 169 Mo par processus. Pour chaque étape de sauvegarde, la valeur mesurée est le temps d'exécution de la reconfiguration « sauvegarde » (du prologue à l'épilogue) sur le processus maitre.

Nombre de serveurs de sauvegarde	Placement		
	Dans l'ordre	Aléatoire	Par *switch*
12	83,07 s	19,71 s	15,56 s
24	41,50 s	16,30 s	7,70 s
60	22,23 s	15,05 s	3,10 s

Tab. 7.1: Durée d'une sauvegarde d'un volume de données de 20 Go en fonction du nombre de serveurs de sauvegarde et du placement des serveurs de sauvegarde

Le tableau 7.1 donne la durée moyenne de la sauvegarde pour les trois placements proposés et pour un nombre variable de serveurs de sauvegarde. Une quinzaine de mesures a été réalisée dans chaque cas, les écarts types sont de l'ordre de 1 %. La figure 7.4 montre ces résultats sous forme d'un diagramme en bâtons.

Cette expérience montre que les performances de la sauvegarde peuvent dépendre fortement du placement des serveurs de sauvegarde au sein de la grappe de calcul. Pour être efficace et lorsque le volume de données à sauvegarder est important, il faut placer

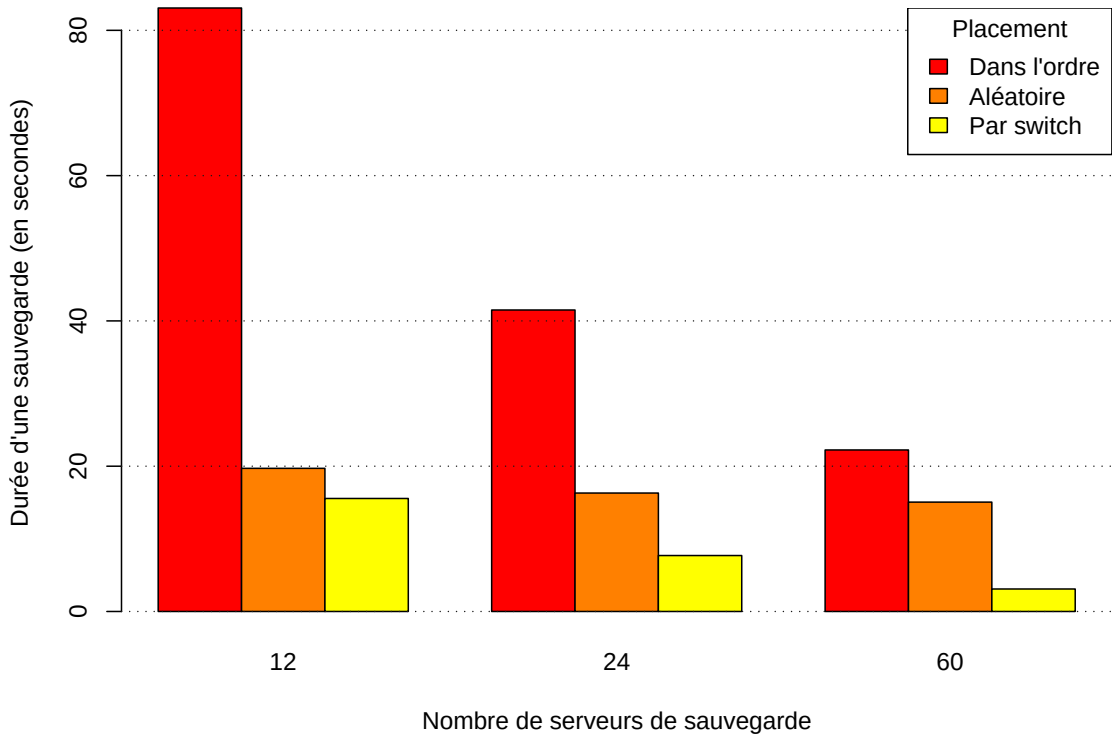


FIG. 7.4: Durée d'une sauvegarde d'un volume de données de 20 Go en fonction du nombre de serveurs de sauvegarde et du placement des serveurs de sauvegarde

les serveurs de sauvegarde au plus près des machines de calcul. Nos résultats montrent une différence d'un facteur 4 entre le placement par *switch* et le placement naïf dans l'ordre. Le placement aléatoire offre un compromis intéressant lorsque la topologie du réseau de communication n'est pas connue.

Dans le cas du placement dans l'ordre, le point de congestion se situe au niveau des liens de second niveau. En effet, ces liens offrent un débit de 3 Gb/s pour des communications entre des groupes de 15 machines, tandis que les liens du premier niveau offrent un débit de 1 Gb/s pour des communications associées à une seule machine. Du point de vue d'une grille de calcul par rapport aux grappes qui la composent, le phénomène est le même.

Enfin, la durée d'une sauvegarde diminue en fonction du nombre de serveurs de sauvegarde. En effet, l'ajout de serveurs de sauvegarde permet d'augmenter la capacité globale de réception du système et de détourner une partie des données sauvegardées passant par le lien limitant vers les nouveaux serveurs ajoutés. Il faut noter que dans notre cas, *i.e.* pour les grilles de calcul, les machines utilisées pour les serveurs de sauvegarde sont les mêmes que celles qui sont utilisées par les processus de calcul. Diminuer le nombre de machines dédiées aux calculs augmentera le temps d'exécution de l'application. Il est donc nécessaire de réaliser un choix sur la proportion de machines à utiliser pour réaliser des serveurs de sauvegarde par rapport aux machines utilisées

pour les processus de calcul.

7.3.3.2 Influence de la taille de la sauvegarde

La durée de la sauvegarde est également influencée par le volume de données à sauvegarder par l'application. Pour mettre en évidence ce phénomène, nous nous plaçons dans des conditions similaires à celles de l'expérience présentée précédemment.

Nous utilisons 132 machines de la grappe d'Orsay de Grid'5000 : 120 machines sont destinées aux processus de calcul et 12 machines sont utilisées pour les serveurs de sauvegarde. Les serveurs de sauvegarde sont placés selon la configuration « par *switch* » décrite à la section précédente.

L'application utilisée est une application itérative de décomposition de domaine. Elle nous permet de fixer la quantité de données sauvegardées par l'application lors de la sauvegarde en choisissant une taille adéquate pour le domaine de calcul.

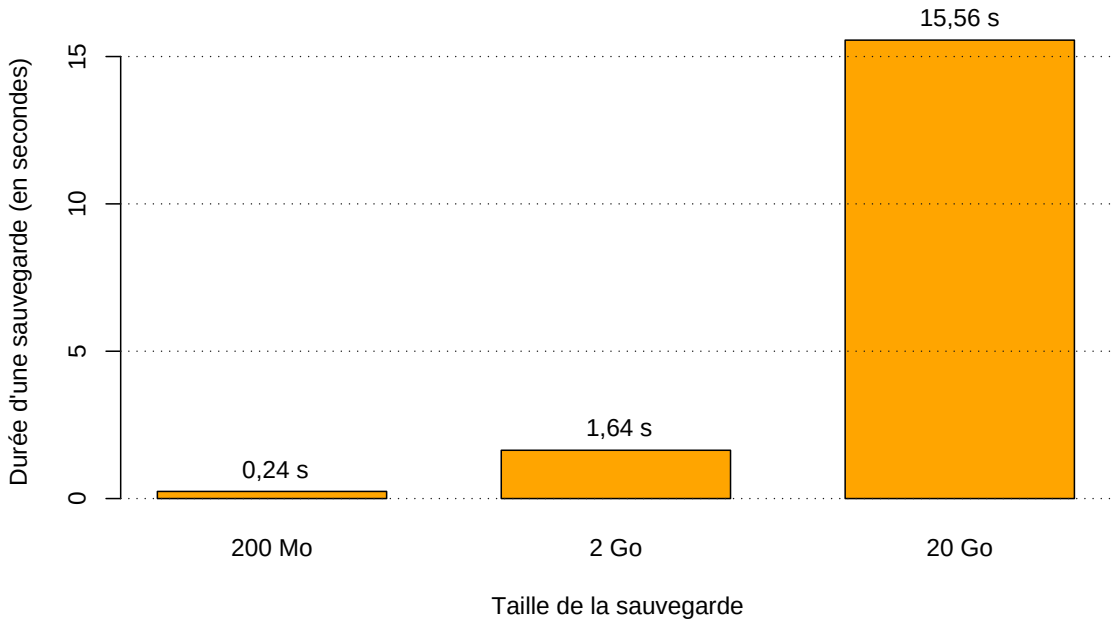


FIG. 7.5: Durée de la sauvegarde en fonction de la taille de la sauvegarde

La figure 7.5 montre la durée moyenne de la sauvegarde pour des tailles de sauvegarde de 200 Mo, 2 Go et 20 Go. Ces mesures sont le résultat d'une quinzaine de mesures et les écarts types sont de l'ordre de 1 %.

Sur les résultats obtenus, une augmentation de la taille de la sauvegarde d'un facteur 10 se traduit également par l'augmentation de la durée de sauvegarde d'un facteur 10. La durée de la sauvegarde est proportionnelle au volume totale de données à sauvegarder[3].

Dans le cas de la sauvegarde de 200 Mo, le débit global de sauvegarde, c'est-à-dire le rapport de la taille de la sauvegarde sur la durée de la sauvegarde, est de 833 Mo/s

[3]Nous rappelons cependant que nous sommes dans un cas où les serveurs de sauvegarde sont correctement placés et les données à sauvegarder sont réparties de manière équilibrée sur les processus.

contre environ 1 250 Mo/s pour les tailles de sauvegarde de 2 Go et 20 Go. Ceci peut s'expliquer par le fait que le temps mesuré prend en compte le cout de la coordination des processus. Pour des grandes tailles de sauvegarde, ce cout est négligeable par rapport au temps de transfert des données. Dans le cas de petites tailles de sauvegarde, cela n'est plus vrai.

7.3.3.3 Influence du nombre de machines

Nous souhaitons également étudier le comportement de notre sauvegarde coordonnée lorsqu'elle est utilisée avec un grand nombre de machines. Pour cela, nous réalisons une expérience qui mesure le temps de sauvegarde de l'application et nous le comparons, dans des conditions identiques, au temps de gestion de la cohérence mutuelle tel qu'il a été étudié dans la section 6.3.

Nous utilisons une application de résolution du problème des N-reines. Cette application compte le nombre de placements possibles de N dames d'un jeu d'échecs sur un échiquier de $N \times N$ cases, sans qu'elles ne se menacent mutuellement. L'algorithme de résolution est récursif et il est écrit en ATHAPASCAN. L'application est exécutée par le moteur d'exécution KAAPI en utilisant un ordonnancement par vol de travail.

Pour cette expérience, nous souhaitons prendre en compte uniquement le cout de la sauvegarde sur un grand nombre de processus et éviter que les mesures ne soient parasitées par d'autres éléments comme par exemple le temps de transfert des données. Pour cela, nous nous plaçons dans les conditions expérimentales suivantes.

- L'application utilisée possède la caractéristique d'avoir un faible volume de données à sauvegarder. La taille maximale constatée durant cette expérience pour la sauvegarde totale de l'application (c'est-à-dire pour tous les processus compris) ne dépassaient pas 14 Mo.
- Nous choisissons de placer les serveurs de sauvegarde au plus proche des processus de calcul. Bien que cela ne corresponde pas à un cas d'utilisation réel, chaque processus de calcul sauvegarde ses informations sur un serveur de sauvegarde localisé sur la même machine.

Ces mesures ont été réalisées sur 879 machines de Grid'5000 de différents sites en utilisant la disposition suivante.

Machines	1–138	139–283	284–403	404–537	538–670	671–748	749–879
Site	Nancy	Sophia	Lille	Bordeaux	Rennes	Toulouse	Lyon

La figure 7.6 donne le temps de sauvegarde mesuré sur l'application des N-reines en fonction du nombre de machines sur lequel est exécutée l'application. La figure montre également, pour des conditions identiques, le temps d'exécution d'une reconfiguration vide avec cohérence mutuelle. Les valeurs affichées sont des moyennes d'environ 90 mesures et les barres d'erreurs représentent l'écart type.

Ces résultats montrent que, lorsque la taille de la sauvegarde est petite, le cout de la sauvegarde coordonnée est presque entièrement lié à la gestion de la cohérence mutuelle. Dans les conditions de cette expérience, la sauvegarde complète de l'application des N-reines s'exécutant sur plus de 800 machines dure environ 1 seconde.

Lorsque le volume de données à sauvegarder devient grand, le cout de la sauvegarde dépend alors plus de la vitesse de transfert des données entre les processus de calcul et

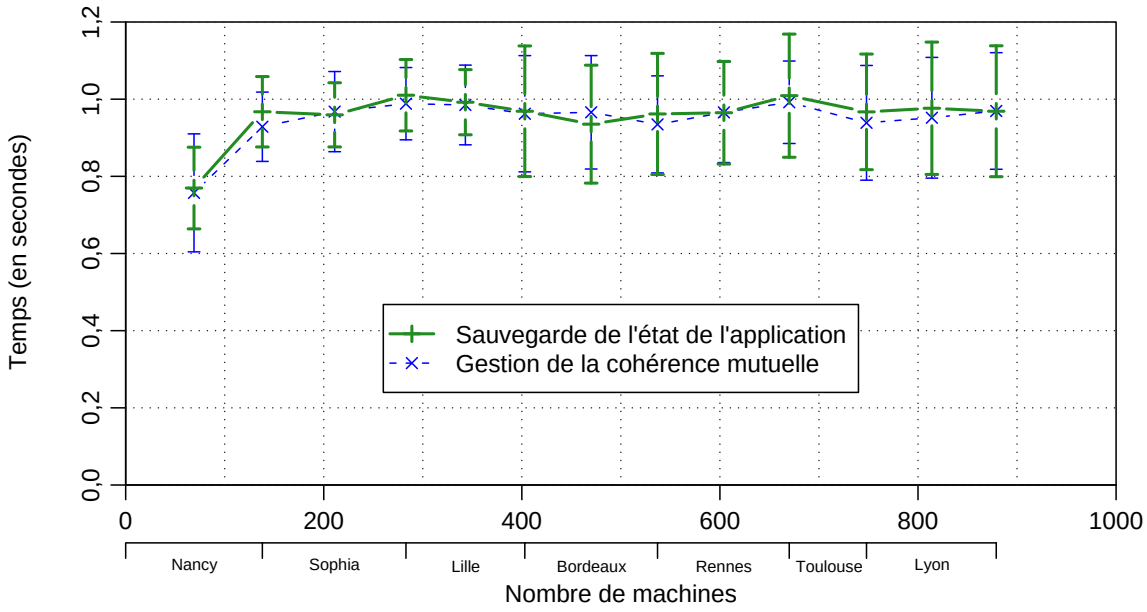


FIG. 7.6: Durée de la sauvegarde en fonction du nombre de machines. La taille totale sauvegardée est au plus de 14Mo.

les serveurs de serveurs. Cette dernière dépend entre autres du réseau de communication utilisé, du nombre de serveurs de sauvegarde et de leur placement.

7.3.3.4 Influence de la période de sauvegarde

Par cette expérience, nous cherchons à étudier l'influence de la sauvegarde sur le temps d'exécution d'une application. Nous utilisons une application itérative de résolution du problème de Poisson par décomposition de domaine. Cette application présente la particularité d'avoir un état sauvegardé de taille constante au cours du temps. La durée de la sauvegarde est donc quasiment la même à chaque étape.

Nous exécutons cette application sur un ensemble de 786 machines de Grid'5000 avec 86 serveurs de sauvegarde. Le temps d'exécution mesuré pour 100 itérations de cette application est de 185,4 secondes. Ce temps sert de référence pour évaluer le surcout engendré par les sauvegardes. La durée moyenne d'une sauvegarde est de 18,7 secondes et la taille de chaque sauvegarde de l'application complète représente 12 Go.

Les résultats de cette expérience sont représentés sur la figure 7.7. Le temps de sauvegarde correspond au temps cumulé par l'application pour effectuer ses sauvegardes, tandis que le temps de calcul indique le temps cumulé à s'exécuter normalement, *i.e.* en dehors des sauvegardes. Les mesures ont été réalisées pour 0 (temps de référence), 2, 4 et 10 sauvegardes durant les 100 itérations de l'exécution de l'application.

La majorité du surcout des exécutions avec sauvegardes est lié au temps passé à sauvegarder les données de l'application. Pour cette application, ce surcout est assez facilement prévisible puisqu'il correspond à la durée d'une étape de sauvegarde multipliée

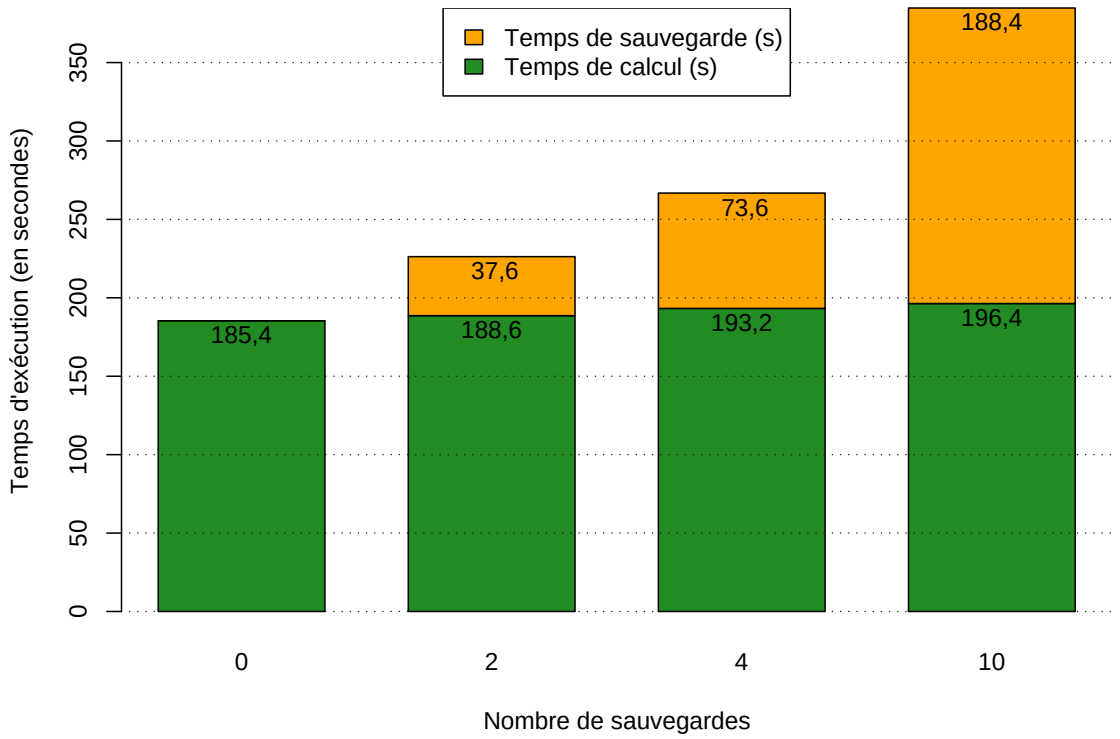


FIG. 7.7: Temps d'exécution de l'application en fonction du nombre de sauvegardes sur 786 machines de Grid'5000 en utilisant au total 86 serveurs de sauvegarde (soit en moyenne 9 machines pour 1 serveur)

par le nombre de sauvegardes réalisées. Ceci est principalement dû au fait que la durée d'une sauvegarde est prévisible car le volume de données sauvegardées à chaque étape est constant.

Nous remarquons également que le temps de calcul augmente légèrement avec le nombre de sauvegardes réalisées. Sans avoir précisément recherché la cause de ce phénomène, nous conjecturons qu'il est provoqué par les perturbations provoquées par chaque sauvegarde sur la charge du réseau et sur l'ordonnancement des calculs. Cependant, cette augmentation reste négligeable par rapport au surcout induit par la sauvegarde proprement dite.

Du fait du temps nécessaire pour garantir la cohérence mutuelle des processus et pour sauvegarder l'état de l'application, le protocole de sauvegarde coordonnée induit un surcout par rapport à une exécution normale d'une application, même sans panne. Connaitre ce surcout et arriver à le prédire permet de déterminer la période de sauvegarde optimale à utiliser durant l'exécution [163, 41]. Cependant, il est nécessaire de connaitre les caractéristiques de l'application, par exemple être capable de prédire le volume de données sauvegardées.

7.3.4 Améliorations possibles

Plusieurs solutions sont possibles pour améliorer les performances de ce protocole de sauvegarde coordonnée bloquante. Nous proposons ici des améliorations pour chacun des trois aspects suivants : la coordination des processus, le volume des données à sauvegarder et la durée de l'étape de sauvegarde locale.

Sauvegarde entre itérations. La sauvegarde entre itérations s'applique uniquement au cas d'une application qui s'exécute par ordonnancement statique. Elle permet de réduire la coordination nécessaire entre les processus[4] en utilisant la synchronisation implicite qui est présente dans le mécanisme d'exécution par ordonnancement statique. En effet, cette synchronisation garantit les propriétés de cohérence et d'accessibilité et peut donc permettre d'éviter l'appel aux fonctions de gestion de la cohérence mutuelle `acquire_mutual_consistency()` et `release_mutual_consistency()`. Ceci permet de réaliser une version spécialisée de la sauvegarde coordonnée qui ne peut sauvegarder l'état de l'application qu'entre les itérations. De plus, cette sauvegarde entre itérations permet également de réduire la quantité de données sauvegardées puisque les données temporaires internes à chaque itération n'ont donc pas besoin d'être sauvegardées.

Sauvegarde incrémentale. Le principe de la sauvegarde incrémentale est de ne sauvegarder que les données qui ont été modifiées depuis la dernière sauvegarde. Ceci permet de réduire le volume de données à sauvegarder car toutes les données ne sont pas forcément modifiées entre chaque étape de sauvegarde. Généralement la sauvegarde incrémentale utilise le bit de modification des pages mémoires des mécanismes de gestion de mémoire virtuelle pour connaitre les données qui ont été modifiées depuis la dernière sauvegarde [118].

Dans Kaapi, nous proposons une solution dont le grain est la donnée partagée Athapascan. En effet, en analysant le graphe de flot de données, il est possible de déterminer les données qui ont été modifiées par le calcul : ce sont au plus celles qui ont un accès en écriture ; les données avec seulement un accès en lecture pourraient n'être sauvegardées qu'une seule fois pour toute l'exécution. Le volume des données à sauvegarder peut alors être réduit.

Émission asynchrone de la sauvegarde. La sauvegarde locale de l'état du processus est une opération bloquante du protocole de sauvegarde coordonnée puisque le calcul ne peut pas reprendre tant que la sauvegarde locale n'est pas terminée au risque de corrompre l'état sauvegardé. En particulier, le cout de cette sauvegarde locale peut être important si le volume de données représentant l'état du processus est grand et si ces données sont émises par le réseau qui constitue alors un goulot d'étranglement.

La solution à ce problème consiste à effectuer une copie locale, synchrone et rapide, de l'état puis d'effectuer l'émission, plus lente, des données de manière asynchrone vers la mémoire stable. La sortie du point local de reconfiguration peut alors s'effectuer dès la fin de la copie pour permettre la reprise du calcul au plus tôt. Sur certains systèmes, il est possible d'implémenter ce mécanisme de manière efficace en utilisant l'appel système

[4]Bien que ce cout de coordination soit relativement faible comme cela a été montré par les expériences de la section 6.3.

`fork` qui effectue une copie paresseuse[5] de l'espace mémoire du processus. Le processus père peut alors continuer les étapes de la sauvegarde tandis que le processus fils réalise la sauvegarde distante de son état. Ceci permet de sauvegarder l'état du processus tel qu'il était au moment de l'appel `fork` et permet de reprendre le calcul plus rapidement car la durée de l'étape de sauvegarde locale est réduite.

Pour résumer, cette optimisation réduit la durée de chaque sauvegarde locale et ainsi le temps pendant lequel chaque processus ne calcule pas. Cependant, le processus coordinateur doit toujours attendre la réception de tous les acquittements pour valider la sauvegarde. Ceci n'est pas gênant puisque cette attente est réalisée dans un processus léger dédié et que pendant ce temps le calcul peut continuer.

7.4 Conclusion

Ce chapitre a présenté les mécanismes nécessaires à la mise en place de la tolérance aux fautes dans un intergiciel et, plus particulièrement dans le cas du moteur d'exécution KAAPI. Nous avons vu comment ces composants sont organisés et interagissent entre eux.

Bien que tous ces composants aient leur importance pour obtenir un mécanisme de tolérance aux fautes efficace et performant, nos travaux portent essentiellement sur les protocoles de sauvegarde et de reprise.

Le reste de ce chapitre s'est porté sur la sauvegarde coordonnée. Nous avons explicité l'état global d'une application KAAPI et présenté le protocole de sauvegarde coordonnée qui repose sur les mécanismes de reconfiguration de la partie II. En particulier, l'implémentation de la sauvegarde coordonnée est simplifiée puisqu'elle est vue comme une reconfiguration avec cohérence mutuelle de l'ensemble des processus de l'application, qui utilise la sauvegarde locale de l'état d'un processus comme fonction de reconfiguration.

Enfin, nous avons mené une série d'expériences qui ont permis de mettre en évidence les couts d'un tel mécanisme de sauvegarde coordonnée. Il en ressort principalement que ce cout est dépendant du nombre et du placement des serveurs de sauvegarde. La taille des données est un paramètre majeur puisque la durée de la sauvegarde est généralement proportionnelle à la quantité de données sauvegardées. Grâce aux mécanismes de reconfiguration de la partie II, le cout de la coordination reste faible même pour un millier de machines. De plus, dès lors que la taille des données à sauvegarder est suffisamment grande, le temps de coordination devient négligeable comparé au temps de transfert des données.

C'est pourquoi les meilleurs moyens pour améliorer les performances de la sauvegarde consistent à réduire le volume des données sauvegardées (grâce à des techniques comme la sauvegarde incrémentale ou la compression) ou à améliorer le temps de transfert entre les processus de calcul et les serveurs de sauvegarde.

La sauvegarde coordonnée présentée dans ce chapitre est utilisée par les techniques de reprise globale et de reprise partielle qui consistuent la suite de nos travaux. Ces deux techniques sont présentées respectivement dans les chapitres suivants 8 et 9.

[5] Les pages mémoires sont partagées entre les deux processus et la copie n'est réellement effectuée que lorsqu'un des deux processus modifie sa mémoire.

Reprise globale 8

Sommaire

8.1 Introduction

Ce chapitre présente le protocole de reprise globale qui est une technique classique qui permet de reprendre l'exécution après une panne. Cette reprise est utilisée en conjonction avec la sauvegarde coordonnée présentée dans le chapitre précédent à la section 7.3.

Dans la section suivante, nous présentons rapidement l'intégration de ce protocole dans Kaapi et la problématique liée à l'utilisation de machines de rechange pour le redémarrage. Nous utiliserons tout au long de ce chapitre une application de décomposition de domaine qui résout un problème de Poisson en trois dimensions (3D) par une méthode aux différences finies. Cette application est représentative, de part son schéma de communication et des volumes de données, de beaucoup d'applications en calcul scientifique.

La section 8.3 présente une modélisation du processus de reprise. La section 8.4 étudie les effets de la sur-décomposition sur la vitesse d'exécution à la reprise. Enfin, le principe de sur-décomposition est mis en pratique avec la technique de reprise globale dans les expériences présentées dans la section 8.5.

8.2 Protocole de reprise globale

La reprise globale est la méthode de reprise classique associée à la sauvegarde coordonnée. Elle est présentée à la section 2.4.2.2. Le principe de cette méthode est

simplement de recharger l'état de l'application tel qu'il a été sauvegardé lors de la dernière étape de sauvegarde. Ainsi tous les processus reprennent le calcul au moment de la dernière sauvegarde.

Nous abordons maintenant deux aspects liés à la reprise : sa cohérence et l'utilisation de machine de remplacement.

Cohérence de la reprise. La cohérence de cette reprise globale repose sur le fait que la sauvegarde coordonnée enregistre un état global cohérent de l'état de l'application. La cohérence de l'état global de l'application est garantie au moment de la sauvegarde, la reprise peut donc s'effectuer de manière transparente.

Dans notre cas, nous utilisons une sauvegarde coordonnée bloquante présentée à la section 7.3. L'état global sauvegardé vérifie à la fois la propriété de cohérence et la propriété d'accessibilité de l'état. Comme expliqué dans le chapitre 5, l'accessibilité signifie que les canaux de communication sont vides. La reprise globale nécessite donc uniquement de rétablir l'état des processus.

Machines de rechange. Les méthodes classiques de reprise globale nécessitent généralement des machines de rechange pour remplacer les machines défaillantes. Ceci est en réalité une contrainte liée à la méthode de sauvegarde de l'état local et au moteur d'exécution.

En effet, lorsque l'état d'un processus est sauvegardé au niveau système, c'est-à-dire sous forme d'un espace mémoire (*cf* section 2.5.1), le seul moyen de rétablir l'état du processus est de recréer le processus. Bien qu'il soit possible de placer plusieurs processus de calcul sur la même machine, cela peut provoquer un déséquilibre de charge qui ralentira l'exécution après la reprise. C'est pourquoi, ces méthodes suggèrent l'utilisation d'une machine de rechange.

Alternativement, si l'état des processus est sauvegardé au niveau de l'intergiciel, c'est-à-dire sous forme d'une représentation abstraite (*cf* section 2.5.1), il est alors possible de recharger l'état de plusieurs processus dans un même processus. Couplé à un algorithme d'ordonnancement capable de rééquilibrer la charge de calcul de l'application, cette solution permet de se passer de machines de rechange.

Puisque Kaapi possède ces deux caractéristiques (une sauvegarde de l'état sous forme d'une représentation abstraite et des algorithmes d'ordonnancement capables de rééquilibrer la charge), nous choisissons de nous intéresser à la reprise sans l'utilisation de machine de rechange. Nous motivons de plus ce choix par son aspect pragmatique. En effet, pour utiliser des machines de rechange pour la reprise, il est nécessaire d'avoir réservé des machines à cet effet qui auraient pu soit servir à cette application ou à d'autres, ou bien d'attendre la réparation des machines défaillantes ou la disponibilité d'autres machines.

8.3 Modélisation de la reprise

Pour évaluer le cout de la reprise globale, nous modélisons le processus de reprise pour différencier les différentes étapes qui la compose. Cette modélisation est illustrée

sur la figure 8.1.

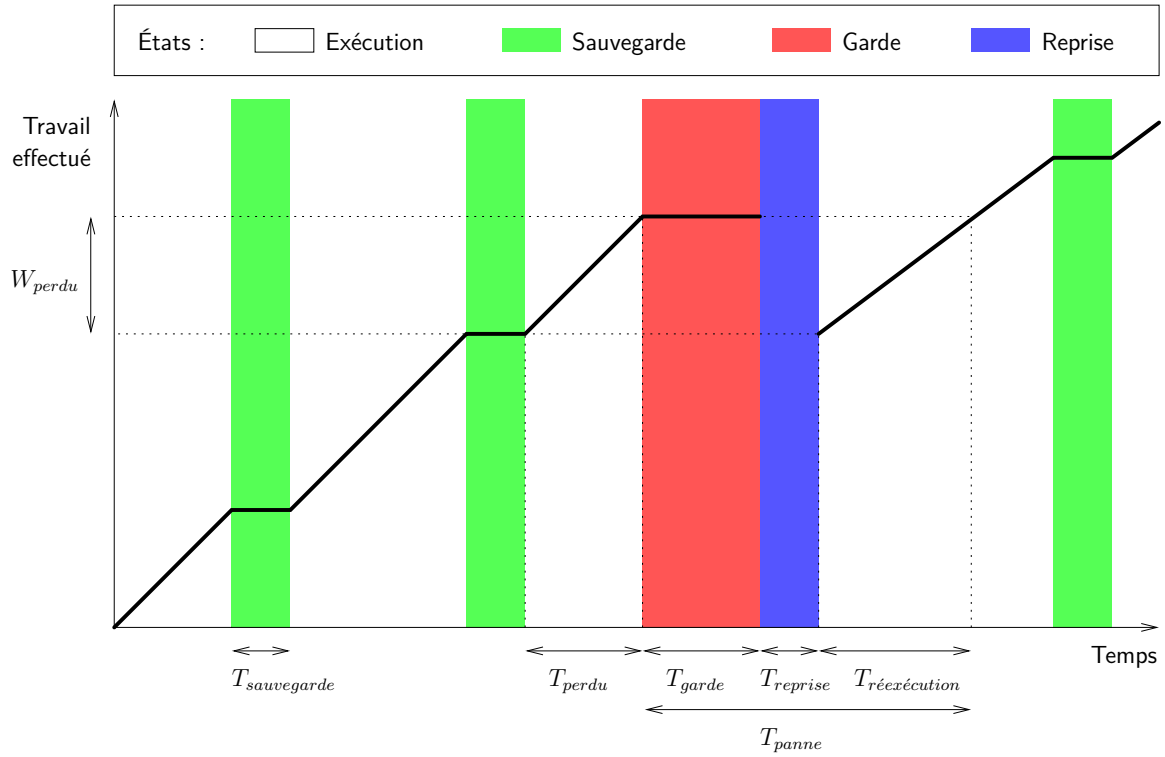


FIG. 8.1: Modélisation de la reprise après une panne

Durant l'exécution, le moteur d'exécution peut prendre quatre états différents. En fonctionnement sans panne, le moteur d'exécution est soit dans l'état Exécution, soit dans l'état Sauvegarde. L'état Exécution signifie que le moteur d'exécution exécute le travail de l'application. L'état Sauvegarde correspond à la période où le calcul est arrêté pour réaliser une sauvegarde de l'état de l'application. Lorsque le moteur d'exécution détecte une panne, il passe dans l'état Garde. Il demeure dans cet état tant que le délai de garde n'est pas passé (*cf* section 7.2.2). L'exécution du protocole de reprise correspond à l'état Reprise.

La courbe de la figure 8.1 montre l'évolution du travail effectué par l'application selon les différents états du moteur d'exécution. Le calcul de l'application ne progresse que lorsque le moteur d'exécution est dans l'état Exécution. En effet, durant la sauvegarde, le calcul est arrêté pour réaliser la sauvegarde de l'état de l'application. De même, l'apparition d'une panne empêche le calcul de progresser puisqu'un processus (et donc une partie des données) a disparu. À la fin de la reprise globale, l'état de l'application de la dernière sauvegarde est rétabli. La quantité de travail effectuée redescend donc au niveau de celui de la dernière étape de sauvegarde.

Grâce à cette modélisation, nous pouvons identifier les grandeurs importantes que nous utiliserons pour évaluer le protocole de reprise globale. Ces grandeurs sont indiquées sur la modélisation proposée à la figure 8.1.

- Le **temps de sauvegarde** est noté $T_{sauvegarde}$. Dans notre modèle, il correspond au temps d'arrêt du calcul lors de la sauvegarde.
- La **quantité de travail perdu**, notée W_{perdu}, correspond au travail qui a été

exécuté par l'application entre la dernière sauvegarde et la panne. Ce travail est perdu puisqu'il a déjà été exécuté mais qu'il devra être réexécuté pour terminer l'exécution de l'application.
- Le **temps perdu**, noté T_{perdu}, correspond au temps qui a été nécessaire pour exécuter le travail perdu.
- T_{garde} est le **temps de la garde** pendant lequel le moteur d'exécution attend la détection d'autres pannes. Ce temps correspond au temps entre la détection de la panne et la décision de la reprise. Il correspond au délai de garde si aucune autre panne ne se produit durant cette période.
- Le **temps de reprise**, noté $T_{reprise}$, est le temps nécessaire pour exécuter le protocole de reprise, en particulier pour recharger l'état sauvegardé de l'application.
- Le **temps de réexécution du travail perdu** est noté $T_{réexécution}$. Il correspond au temps d'exécution nécessaire pour que l'application retrouve la même quantité de travail qu'elle avait juste avant que la panne se produise.

Nous notons alors T_{panne}, le surcout induit par la panne comparé à une exécution sans panne. Dans notre modèle, il s'exprime de la manière suivante :

$$T_{panne} = T_{garde} + T_{reprise} + T_{réexécution}$$

Pour limiter l'impact d'une panne sur l'exécution d'une application, il faut réduire T_{panne}, et donc les trois grandeurs T_{garde}, $T_{reprise}$ et $T_{réexécution}$.

La grandeur T_{garde} est liée au délai de garde qui est une valeur configurable. Le délai de garde est indépendant du protocole de reprise et mais il est lié à la qualité des détecteurs de pannes. Plus les détecteurs de pannes sont réactifs, plus le délai de garde peut être diminué. Avec un délai de garde trop faible, le risque est de lancer le protocole de reprise sans avoir détecté toutes les pannes, ce qui provoquera l'échec le reprise[1]. Pour ces raisons, nous ne nous intéresserons pas à cette valeur et nous la considèrerons comme fixée.

La grandeur $T_{reprise}$ est intrinsèquement liée au protocole de reprise. Ce temps prend en compte le temps nécessaire pour recharger l'état des processus défaillants et le cout de l'algorithme qui reconstruit ou vérifie la cohérence de l'état global obtenu. Une manière de réduire $T_{reprise}$ est d'améliorer les performances de la mémoire stable de manière à rendre les opérations d'interrogation et de recouvrement plus rapides.

Dans ce manuscrit nous nous intéressons principalement à réduire la grandeur $T_{réexécution}$. Dans la suite de ce chapitre, nous étudierons comment réduire cette grandeur pour la reprise globale grâce à la technique de sur-décomposition. Dans ce cas, la quantité de travail perdu W_{perdu} correspond exactement au travail exécuté entre la dernière sauvegarde et la défaillance. Alternativement, nous proposerons dans le chapitre 9 un protocole de reprise partielle qui permet de réduire la quantité de travail perdu W_{perdu}. Ce qui permet grâce à un réordonnancement adéquate de diminuer le temps $T_{réexécution}$ de réexécution du travail perdu.

8.4 Effet de la sur-décomposition

Dans le but d'expliquer l'effet de la sur-décomposition sur la reprise globale, nous proposons d'abord dans cette section une modélisation d'une application cible de notre

[1]Mais une nouvelle reprise peut toujours être relancée.

mécanisme de tolérance aux fautes par sauvegarde coordonnée et reprise. L'application considérée est une application itérative de calcul scientifique de type décomposition de domaine.

Ensuite nous présentons le principe de sur-décomposition d'une application et nous étudions ses effets sur l'exécution avant et après la panne.

8.4.1 Modélisation d'une application

Nous considérons le problème de Poisson sur un domaine de calcul en trois dimensions[2]. L'application calcule la solution de ce problème d'EDP classique par un schéma aux différences finies dont le système linéaire résultant est résolu par la méthode itérative de Jacobi. À chaque itération, chaque élément du domaine de calcul est mis à jour grâce à la valeur de ses voisins. Au fur et à mesure des itérations, le domaine de calcul converge vers la solution du problème de Poisson qui est un point fixe.

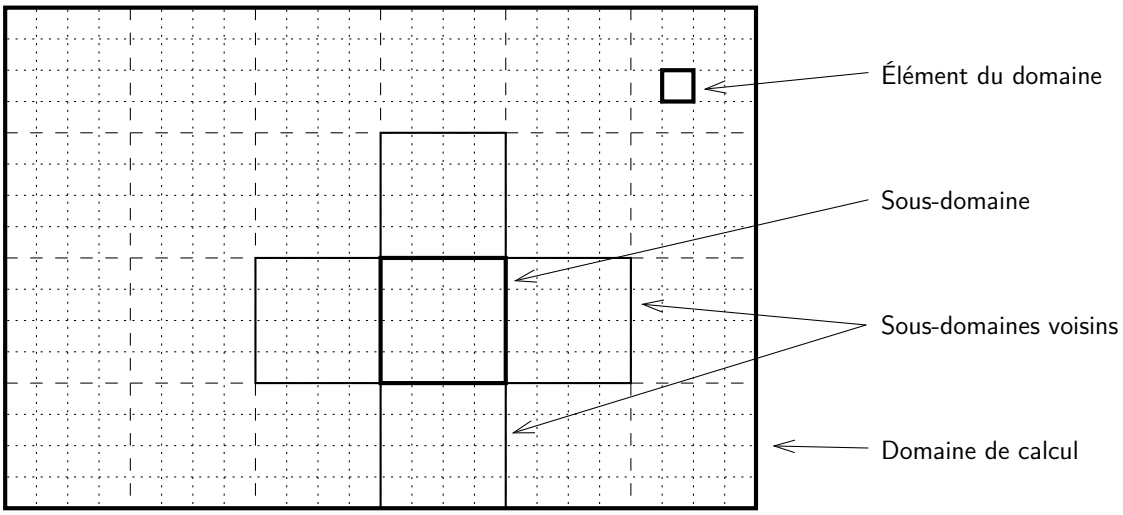


FIG. 8.2: Décomposition d'un domaine de calcul

Le **domaine de calcul** est un ensemble de points ; ces points sont appelés éléments du domaine. Les **éléments du domaine** sont généralement des nombres réels représentés par le type `double`. Le domaine de calcul complet est découpé en d **sous-domaines** qui regroupent chacun plusieurs éléments du domaine. Ces sous-domaines représentent l'unité de base du calcul. Nous définissons les **voisins d'un sous-domaine** comme les sous-domaines adjacents au sous-domaine considéré qui interviennent dans le calcul pour la mise à jour du sous-domaine. Ces définitions sont illustrées à la figure 8.2.

Le graphe de flot de données ATHAPASCAN de l'application de résolution selon la méthode de Jacobi est défini à partir de la décomposition en sous-domaine du domaine de calcul. Ainsi, à chaque sous-domaine de calcul est associée une donnée partagée ATHAPASCAN. À chaque itération du calcul, la mise à jour du domaine complet est réalisée par la mise à jour de chaque sous-domaine par l'intermédiaire d'une tâche de calcul ATHAPASCAN[3]. Cette tâche de calcul dépend du sous-domaine à mettre à jour

[2]Pour des soucis de clarté, les figures présenterons un domaine en une ou deux dimensions.

[3]Ceci correspond juste à une description simplifiée, la vraie application est plus complexe.

et de la valeur des sous-domaines voisins à l'itération précédente. Le graphe de flot de données résultant est donné à la figure 8.3.

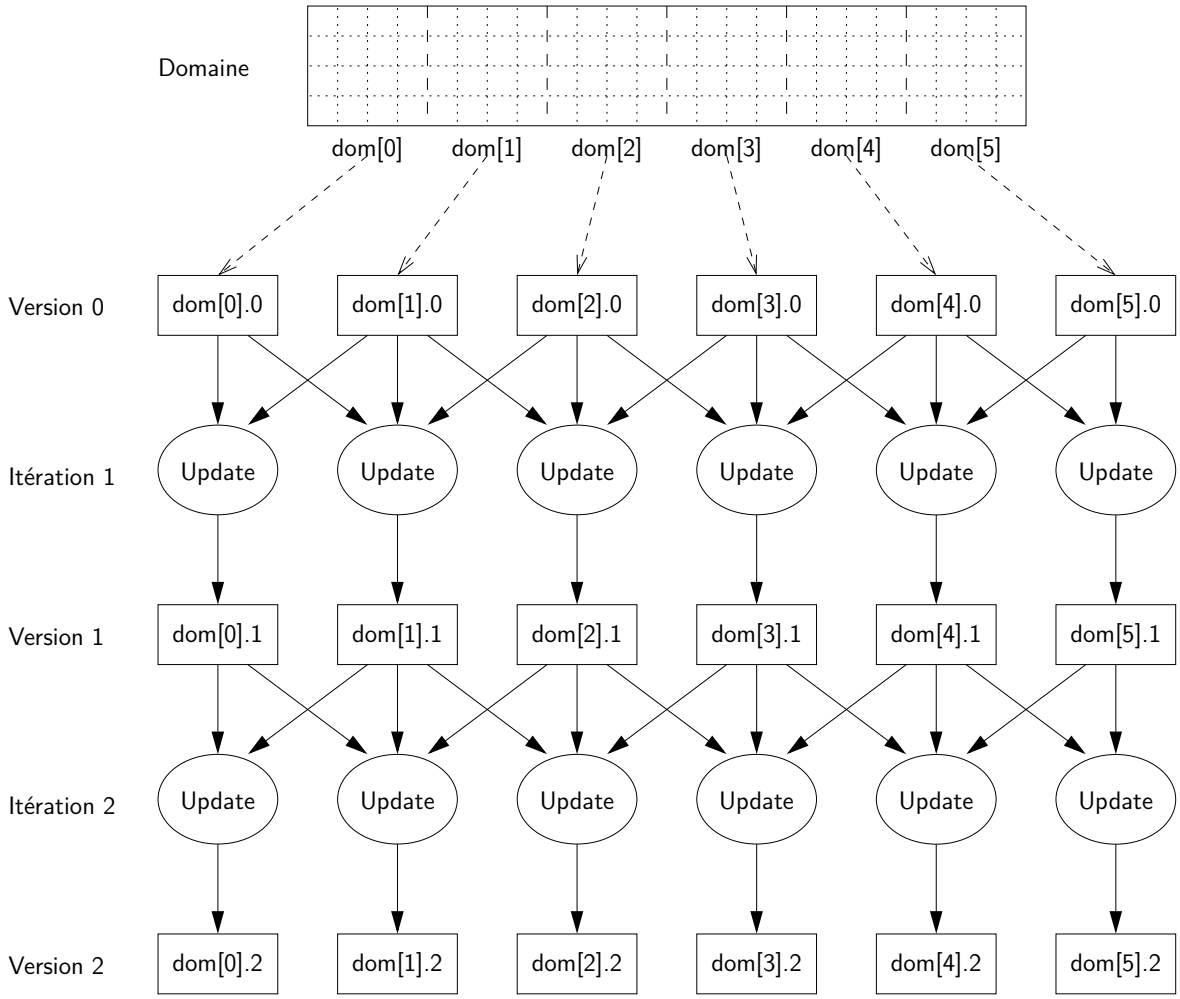


Fig. 8.3: Graphe de flot de données simplifié d'une méthode de Jacobi sur quatre sous-domaines. Le graphe représentent les dépendances d'un problème de Poisson en une dimension (1D) représenté en haut de la figure.

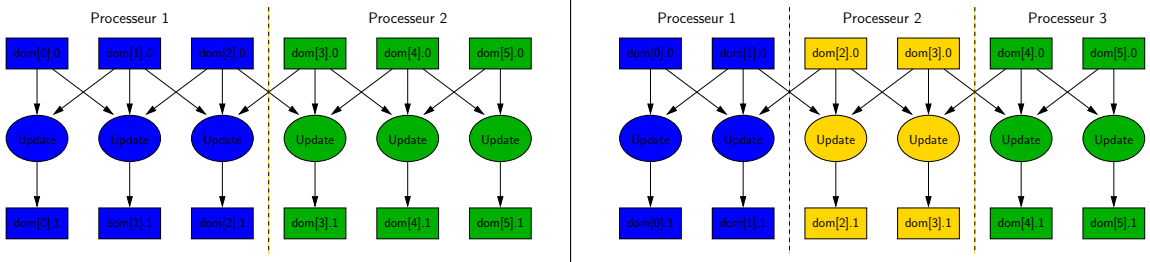
8.4.2 Sur-décomposition

L'approche la plus simple pour paralléliser une application par décomposition de domaine sur n processeurs est d'utiliser une découpe en n sous-domaines. Chaque sous-domaine est alors affecté à un processeur et, inversement, chaque processeur ne travaille que sur un sous-domaine. Cette approche classique est couramment utilisée, notamment avec le modèle de programmation MPI. En effet, le code est plus simple car il ne nécessite pas de gérer plusieurs sous-domaines par processeur.

Cette approche présente l'inconvénient de lier le découpage, et donc la parallélisation au nombre de processeurs disponibles pour exécuter le calcul. L'équilibrage de charge est réalisé au moment de la découpe et il est donc dépendant du nombre de processeurs sur

lequel l'application s'exécute. Il peut difficilement être remis en cause durant l'exécution, en particulier si le nombre de processeurs disponibles change.

La **sur-décomposition** est une approche qui vise à découper le domaine de calcul en un nombre de sous-domaines d très supérieur au nombre n de processeurs. L'objectif est alors d'avoir une découpe en sous-domaines qui peuvent être placés plus librement sur les processeurs. On dit alors qu'elle est indépendante du nombre de processeurs utilisés pour l'exécution. L'ensemble des sous-domaines sont alors repliés sur l'ensemble des processeurs disponibles. Ceci est réalisé dans KAAPI grâce au mécanisme de partitionnement statique présenté à la section 4.3.2.3. Un exemple de regroupement est illustré à la figure 8.4.



(a) Regroupement sur 2 processeurs

(b) Regroupement sur 3 processeurs

FIG. 8.4: Regroupement des calculs après sur-décomposition : le parallélisme de l'application est décrit de manière indépendante du nombre de processeurs. Dans ce cas particulier, le choix du nombre de sous-domaines et du nombre de processeurs fait que chaque processeur reçoit un même nombre de tâches.

Le sous-domaine correspond donc au grain de calcul, c'est-à-dire celui qui est utilisé pour réaliser l'ordonnancement. Pour un domaine de calcul fixé, un plus grand nombre de sous-domaines implique un temps de calcul par sous-domaine plus petit. On appellera **grain** ce temps d'exécution d'une tâche sur un sous-domaine. À cela deux conséquences : le nombre de tâches qui composent l'application est plus important mais ces tâches ont un grain plus faible. La charge peut être équilibrée plus facilement sur un nombre quelconque de processeurs.

La sur-décomposition est présentée ici car elle présente de bonnes propriétés vis-à-vis d'une exécution en présence de fautes. Elle offre en particulier plus de flexibilité sur l'ordonnancement lorsque le nombre de processeurs varie, à cause des pannes, et qu'il n'y a pas de machines de rechange. Ces bonnes propriétés de la sur-décomposition sont présentées dans les sections suivantes.

8.4.3 Exécution avant et après une panne

Nous considérons l'application modélisée à la section 8.4.1 et nous allons évaluer le temps d'exécution d'une itération selon la décomposition utilisée. Pour cette évaluation, nous choisissons d'ignorer les communications et de prendre uniquement en compte les calculs. Nous supposons également que les processeurs sont homogènes.

Pour cette application et pour un domaine de calcul fixé, nous notons W le travail correspondant à la mise à jour du domaine complet. W représente donc le travail effectué

durant une itération de l'application.

Nous notons également T_n^d le temps d'exécution du travail W découpé en d sous-domaines sur n processeurs. T_1^1, également noté T, correspond donc au temps d'exécution séquentiel sans découpage du domaine de calcul. Pour une exécution sur n processeurs, le temps d'exécution optimal (*i.e.* le plus petit) correspond à une découpe en n sous-domaines et il est noté T_n^n.

Avant une panne. De manière générale, pour une découpe en d sous-domaines, le temps d'exécution de chaque sous-domaine est $\frac{T}{d}$. De plus, la modélisation de l'application de la section 8.4.1 nous permet de considérer qu'au sein d'une itération, les tâches de calcul associées à chaque sous-domaine peuvent être exécutées de manière indépendante.

Il en découle que si les d sous-domaines sont répartis de manière équilibrée sur n processeurs, on obtient alors un nombre maximal de $\left\lceil \frac{d}{n} \right\rceil$ par processeurs. Le chemin critique de l'exécution d'une itération correspond au processeur auquel est associé le plus de sous-domaines. On obtient un temps d'exécution pour une découpe en d sous-domaines sur n processeurs de $\left\lceil \frac{d}{n} \right\rceil \times \frac{T}{d}$.

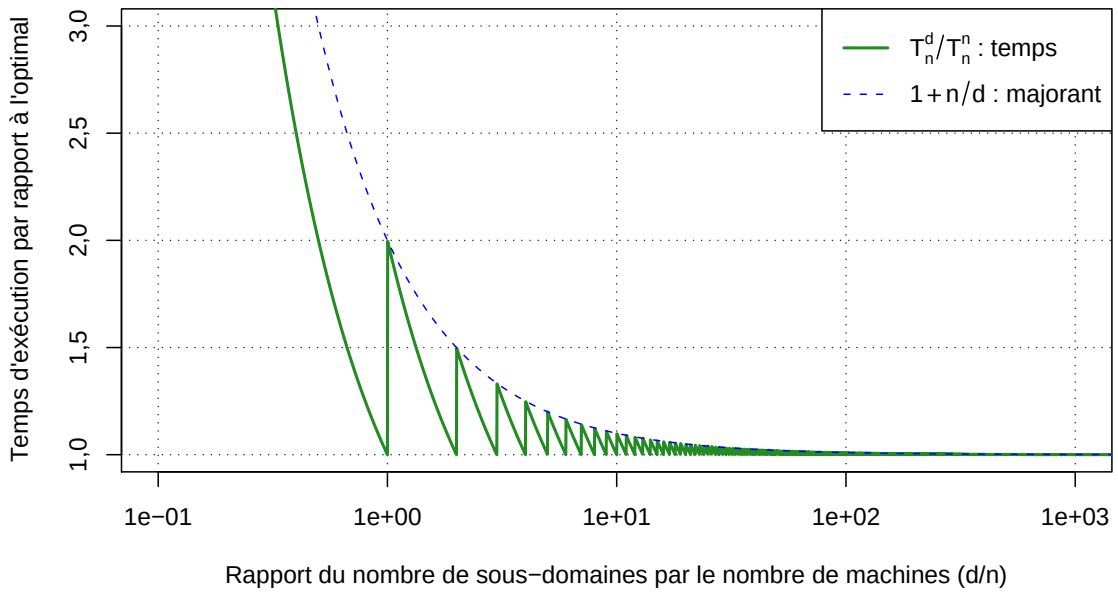


FIG. 8.5: Temps d'exécution sur 1000 machines par rapport à l'optimal en fonction du nombre de sous-domaines d de la décomposition

Pour une décomposition en d sous-domaines et une exécution en n machines, le temps d'exécution est T_n^d ; nous le comparons au temps d'exécution optimal qui est T_n^n.

$$\frac{T_n^d}{T_n^n} = \left\lceil \frac{d}{n} \right\rceil \times \frac{n}{d} \leq 1 + \frac{n}{d}$$

La figure 8.5 montre ce temps d'exécution par rapport à l'optimal en fonction du rapport de sur-décomposition (*i.e.* $\frac{d}{n}$). Pour une sur-décomposition $\frac{d}{n} > 100$, le surcout théorique du temps d'exécution est de moins de 1 %.

Après une panne. Nous étudions maintenant le cas de la panne de p machines et nous supposons qu'il y a pas de machine de rechange disponible. L'exécution peut continuer mais seulement sur $n - p$ machines. Dans ce cas là, l'optimal est T^{n-p}_{n-p}.

Le protocole de reprise globale permet de redémarrer l'application mais le découpage utilisé avant la panne reste le même. La perte des machines défaillantes provoque un déséquilibre de la charge. Pour une décomposition exacte en n, le travail peut difficilement être rééquilibré après la panne puisque l'unité de calcul est trop grosse. Dans le cas d'une sur-décomposition, le grain de calcul est suffisamment petit pour permettre un rééquilibrage proche de l'optimal.

	Avant la panne, exécution sur n machines	Après la panne, exécution sur $n - p$ machines
Décomposition exacte en n	$T^n_n = \frac{T}{n}$	$T^n_{n-p} = \left\lceil \frac{n}{n-p} \right\rceil \times \frac{T}{n}$
Décomposition en d	$T^d_n = \left\lceil \frac{d}{n} \right\rceil \times \frac{T}{d}$	$T^d_{n-p} = \left\lceil \frac{d}{n-p} \right\rceil \times \frac{T}{d}$

Tab. 8.1: Temps d'exécution en fonction du type de décomposition et du nombre de machines avant et après une panne

Le tableau 8.1 donne un récapitulatif du temps d'exécution d'une itération en fonction des différentes conditions suivant notre modèle.

En prenant en compte les pannes, nous obtenons le rapport du temps d'exécution sur le temps optimal :

$$\frac{T^d_{n-p}}{T^{n-p}_{n-p}} = \left\lceil \frac{d}{n-p} \right\rceil \times \frac{n-p}{d} \leq 1 + \frac{n-p}{d} \leq \frac{n}{d}$$

$T^d_{1000-p}/T^{1000-p}_{1000-p}$	$p = 0$	$p = 1$	$p = 10$	$p = 100$	$p = 500$
$d = 1\ 000$	1	1,998	1,98	1,8	1
$d = 10\ 000$	1	1,0989	1,089	1,08	1
$d = 100\ 000$	1	1,0090	1,0098	1,008	1
$d = 1\ 000\ 000$	1	1,0010	1,0009	1,0008	1

Tab. 8.2: Ratio du temps d'exécution sur $1000 - p$ machines par rapport à l'optimal après la panne de p machines pour une décomposition en d

Le tableau 8.2 donne la valeur du rapport $T^d_{1000-p}/T^{1000-p}_{1000-p}$ qui correspond à une exécution sur 1000 machines. Les valeurs de ce tableau sont représentées dans le

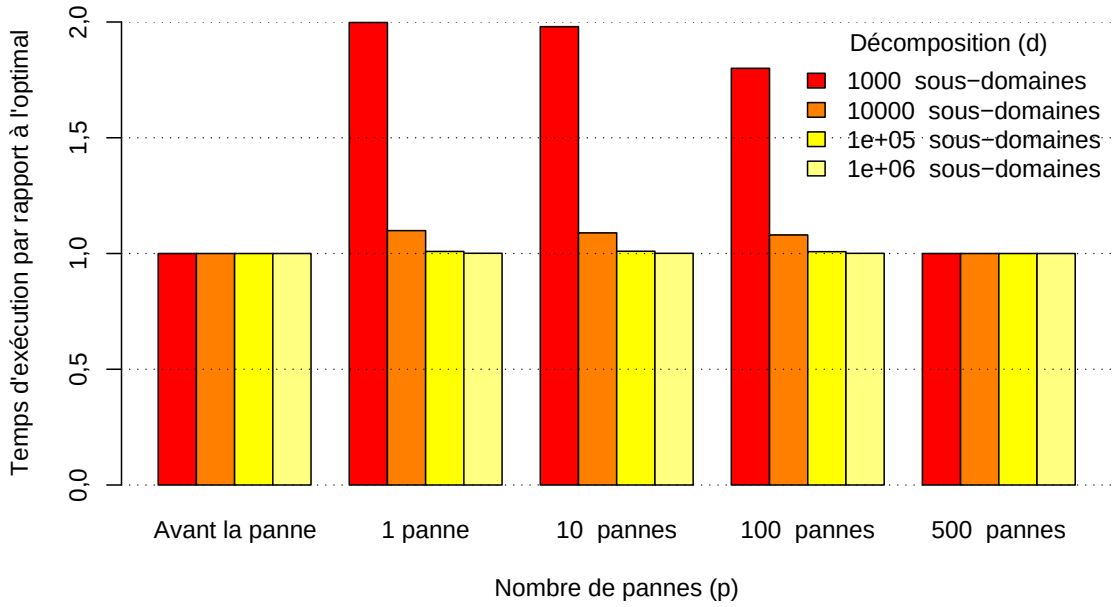


FIG. 8.6: Temps d'exécution sur $1000 - p$ machines par rapport à l'optimal après la panne de p machines

diagramme en bâtons de la figure 8.6. Le cas $d = 1000$ correspond au cas de la décomposition exacte avant la panne. Nous observons que la décomposition exacte offre un temps optimal avant la panne. Cependant, en cas de panne et si les machines défaillantes ne peuvent pas être remplacées, le temps d'exécution est presque à un facteur 2 de l'optimal. La sur-décomposition ($d \gg n$) permet d'avoir un temps d'exécution proche de l'optimal à un facteur $\frac{n}{d}$.

Il est important de constater que ces résultats influencent à la fois le temps de réexécution du travail perdu $T_{réexécution}$ mais également le temps d'exécution de toutes les itérations postérieures à la reprise jusqu'à la venue de machines de rechange. Le gain apporté par la sur-décomposition bénéficie donc à tout le reste de l'exécution.

Nous allons présenter dans la section suivante, les expériences réellement menées sur une telle application et sur des machines de Grid'5000. Nous verrons que l'hypothèse implicite prise ci-dessus, qui considère un surcout de gestion des sous-domaines nul, n'est qu'une première approximation. Ces expériences vont nous permettre de mettre en évidence l'effet de la sur-décomposition sur l'exécution après la reprise.

8.5 Expérimentations

Nous étudions l'influence de la sur-décomposition sur la vitesse d'exécution d'une application pour une exécution sur une ou plusieurs machines. Puis, nous mettons en évidence de manière pratique le gain que la sur-décomposition apporte sur la vitesse

d'exécution après une reprise globale. Enfin, nous détaillons les différents couts engendrés par une reprise globale.

8.5.1 Influence de la sur-décomposition

Pour étudier l'effet de la sur-décomposition sur le temps de calcul d'une application, nous utilisons une application itérative de décomposition de domaine sur un domaine en 3 dimensions. Cette application correspond à la description donnée dans la section 8.4.1. Elle est écrite avec le langage ATHAPASCAN et son extension présentée à la section 4.2.2. Elle est exécutée grâce au moteur d'exécution KAAPI. Cette application présente la caractéristique suivante : si les conditions ne changent pas, le temps d'exécution d'une itération, c'est-à-dire le temps de mise-à-jour de tous les domaines, est constant.

Pour réaliser cette expérience, nous fixons le domaine de calcul sur un processeur à 10^7 nombres réels de type `double`, soit environ 76 Mo. De manière à évaluer la vitesse d'exécution de l'application, nous mesurons le temps d'exécution d'une itération pour différentes décompositions. Ces mesures sont réalisées pour une exécution sur 1 machine et pour une exécution sur 100 machines. De plus, sur chaque machine, un cœur de calcul est utilisé. Dans tous les cas, nous conservons un domaine de calcul par machine fixé à 10^7 réels ; donc pour l'exécution sur 100 machines, le domaine total de calcul est de $100 \times 10^7 = 10^9$ réels, soit environ 7,6 Go.

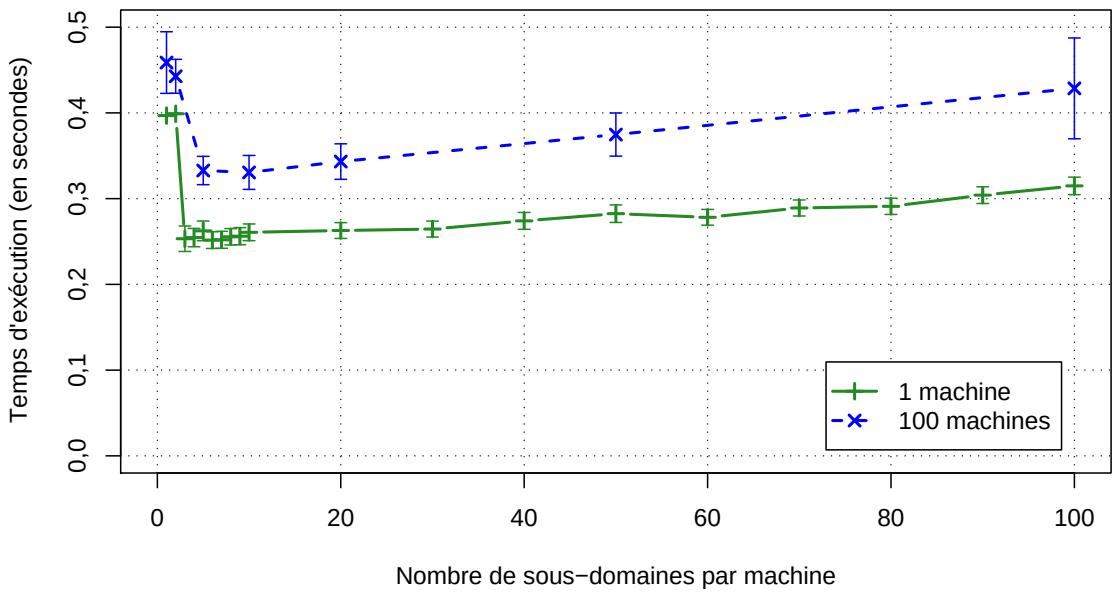


FIG. 8.7: Temps d'exécution d'une itération en fonction de la sur-décomposition utilisée

Les mesures ont été réalisées sur la grappe Griffon du site de Nancy de Grid'5000. La figure 8.7 montre les résultats des mesures. Les valeurs tracées sont la moyenne des temps mesurés pour une centaine d'itérations et les écarts types sont affichés autour de chaque point sous forme de barres d'erreur. L'axe des abscisses indique la décomposition

utilisée pour le calcul : un nombre d de sous-domaines signifie que le domaine de 10^7 réels est découpé en d blocs de $10^7/d$ réels.

Nous nous intéressons tout d'abord à la courbe des temps d'itération sur 1 machine. Le temps d'une itération de mise à jour du domaine de calcul pour une décomposition en 1 ou 2 sous-domaines est d'environ 0,4 secondes. À partir d'une décomposition en 3 sous-domaines, le temps d'exécution diminue d'environ 35 % à 0,25 secondes. En effet, à partir d'un découpage en 3 sous-domaines, la petite taille des blocs permet de réduire les défauts de cache lors de la mise à jour du domaine et donc d'accélérer le calcul.

Ensuite, à partir de 3 sous-domaines, le temps d'une itération augmente faiblement de manière linéaire en fonction du nombre de sous-domaines. Ce surcout lié au nombre de sous-domaines est le cout de gestion du parallélisme. Il correspond aux opérations arithmétiques supplémentaires nécessaires pour permettre la parallélisation (*i.e.* le découpage) tout en garantissant un résultat identique du calcul.

Une partie de ce surcout vient de l'algorithme utilisé pour effectuer le calcul en parallèle (ou en utilisant plusieurs sous-domaines). Par exemple, sur l'algorithme de décomposition de domaine utilisé lors de cette expérience, ces surcouts sont principalement dus à la gestion des frontières de chacun des sous-domaines.

L'autre partie de ce surcout vient du moteur d'exécution utilisé et en particulier de la gestion des structures de données utilisées. Le programme Athapascan correspondant à cet exemple crée un nombre de tâches proportionnel au nombre de sous-domaines utilisés pour le calcul. La gestion de ces tâches a un cout à l'exécution.

Pour la courbe des temps d'itération sur 100 machines, le domaine de calcul par machine est le même que pour l'exécution sur 1 machine. La quantité de calcul par machine est la même. Cependant, les machines doivent communiquer pour échanger les frontières de leur domaine de calcul.

La courbe correspondant à l'exécution sur 100 machines présente la même forme que la courbe sur 1 machine. Cependant, elle est décalée vers le haut. Ce décalage correspond au surcout des communications. Ce décalage est de l'ordre de 0,05 seconde pour un nombre de sous-domaines faible, et il est de l'ordre 0,1 seconde pour 100 sous-domaines par machine.

Nous avons montré expérimentalement le bénéfice que peut apporter la sur-décomposition sur la vitesse d'exécution. Deux phénomènes sont mis en évidence : les effets de cache, qui permettent d'augmenter la vitesse du calcul lorsque la taille des sous-domaines passe en dessous d'un certain seuil, et le surcout de gestion du parallélisme, qui augmente proportionnellement au nombre de sous-domaines utilisés.

Il faut impérativement remarquer que ces phénomènes sont dépendants de nombreux paramètres. L'algorithme de calcul utilisé par l'application, la taille et la forme des domaine et également le découpage ont chacun leur influence.

8.5.2 Influence des pannes

Cette section propose d'étudier l'influence des pannes sur la vitesse d'exécution de l'application en fonction de la sur-décomposition utilisée. Le but de cette expérience est

de se comparer aux résultats présentés à la figure 8.6 obtenus à partir de la modélisation de la section 8.4.

Pour cela, nous utilisons la même application que celle utilisée à la section précédente et les mêmes conditions expérimentales. Nous exécutons l'application sur 100 machines en utilisant un domaine identique : un domaine total de 10^9 réels, soit 10^7 réels par machine.

Pour cette expérience, nous simulons la panne de p machines et redémarrons l'application, sans machine de rechange, sur les $100 - p$ machines restantes. Nous mesurons alors le temps d'exécution d'une itération une fois le redémarrage terminé.

Les mesures sont réalisées pour des décompositions du domaine global (*i.e.* pour 100 les machines) entre 100 et 10000 sous-domaines, et pour des pannes de 0 (*i.e.* avant la panne) et de 1 à 50 machines.

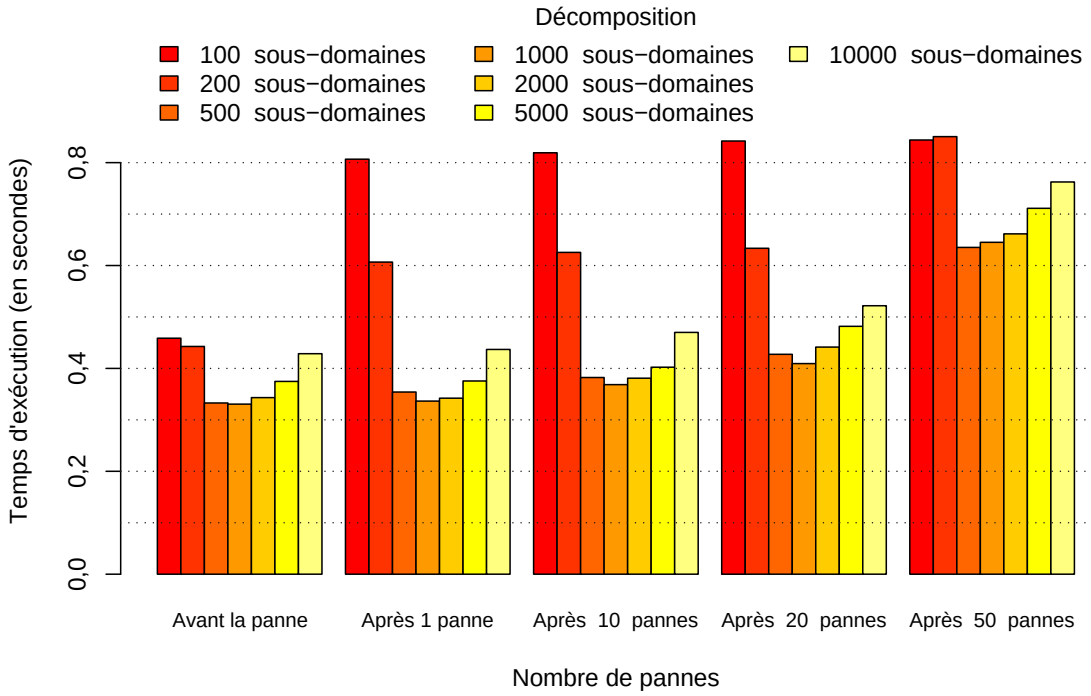


FIG. 8.8: Temps d'exécution d'une itération sur $100 - p$ après la panne de p machines pour différentes sur-décompositions

Le diagramme en bâtons de la figure 8.8 rapporte les résultats des mesures effectuées. Les valeurs affichées correspondent à la moyenne des temps de 200 itérations. Ce diagramme est à comparer avec celui de la figure 8.6 page 168. Cependant, il est important de noter les différences suivantes.

- Le diagramme issu de la modélisation de la section 8.4 représente une exécution sur $1000 - p$ machines et un taux de sur-décomposition de 1 à 1000 fois le nombre de machines. Quant à lui, le diagramme issu des expériences correspond à une exécution sur $100 - p$ machines avec un taux de sur-décomposition de 1 à 100 fois le nombre de machines.
- Le diagramme de la figure 8.8 issu des résultats expérimentaux n'est pas normalisé par rapport au temps d'exécution optimal.

- Enfin, le modèle présenté à la section 8.4 qui est utilisé pour réalisée la figure 8.6 ne prend pas en compte les phénomènes d'effets de cache et de surcout de gestion du parallélisme qui ont été mis en évidence dans la section 8.5.1.

Pour revenir aux résultats expérimentaux de la figure 8.8, nous remarquons tout d'abord que, pour un nombre de pannes fixé, le temps d'exécution pour différentes décompositions suit la même forme que la courbe qui a été présentée dans l'expérience de la section précédente (figure 8.7). Ceci est particulièrement vrai pour les cas où le nombre de sous-domaines par machines est équilibré (0 et 50 pannes).

De même, pour une décomposition fixée, le temps d'exécution d'une itération augmente en fonction du nombre de pannes. Comme le domaine total de calcul reste le même et que le nombre machines participant à l'exécution diminue $(100 - p)$, il y a plus de travail par machine. Ainsi, le temps d'exécution d'une itération après le panne de 50 machines est deux fois plus long que sur 100 machines (*i.e.* avant la panne).

Les cas « Avant la panne » et « Après 50 pannes » sont les cas où le travail est équilibré quelle que soit la décomposition utilisée. Dans les autres cas et pour une décomposition en un faible nombre de sous-domaines par rapport au nombre de machines, nous observons un déséquilibre du nombre de sous-domaines par machines. Ce phénomène apparait dans les mesures par un temps d'exécution plus élevé.

Pour une décomposition en 100 sous-domaines, l'exécution avant la panne (*i.e.* sur 100 machines) se fait avec 1 sous-domaine par machine. Après la panne d'une machine, le travail est déséquilibré : il y a 98 machines avec un 1 sous-domaine et 1 machine avec 2 sous-domaines. Le temps d'exécution étant donné par la machine la plus chargée, on obtient un facteur 2 d'augmentation du temps de calcul, ce qui correspond à peu près aux mesures relevées. Ce facteur est le même pour les cas avec 10 et 20 pannes.

Pour une décomposition en 200 sous-domaines, l'exécution avant la panne se fait avec 2 sous-domaines par machines (mais ces sous-domaines sont deux fois plus petits que dans le cas précédent). Après la panne d'une machine, le travail est également déséquilibré puisque il y a 98 machines avec 2 sous-domaines et 1 machine avec 3 sous-domaines. Dans ce cas, et également dans les cas avec 10 et 20 pannes, le facteur d'augmentation du temps d'itération par rapport l'exécution avant la panne est de 3/2, ce qui correspond également aux valeurs observées.

La modélisation du temps d'exécution d'une itération après la reprise, que nous avons présentée à la section 8.4, ne prend pas en compte les phénomènes liés aux effets de cache et au surcout de gestion du parallélisme. Cependant, il permet tout de même de prédire partiellement les comportements expérimentaux observés.

8.5.3 Temps de réexécution du travail perdu

Nous avons vu précédemment l'impact de la sur-décomposition sur la capacité de bien rééquilibrer la charge après une panne. Dans la section 7.3 du chapitre précédent, nous avons présenté le cout de l'étape de sauvegarde coordonnée.

Pour finir l'évaluation de ce protocole de reprise globale et avant d'enchaîner sur le protocole de reprise partielle, nous allons détailler le cout d'une reprise globale. En reprenant la modélisation de la figure 8.1, nous mesurons les deux éléments suivants :

- $T_{reprise}$ qui est le temps d'exécution du protocole de reprise proprement dit. Dans l'implémentation de ce protocole dans KAAPI, cette étape consiste à contacter les processus non défaillants et à réinitialiser leur état. Une nouvelle distribution du travail est calculée pour tenir compte de la disparation des machines défaillantes en utilisant le partitionnement statique (*cf* section 4.3.2.3). L'ensemble des tâches de calcul est distribué sur tous les processus participant. La redistribution des données n'est pas prise en compte dans le temps $T_{reprise}$ puisque, comme nous l'avons décrit dans 4.3.2.3, la redistribution des données est définie sous forme de tâches de communication ; elle est donc réalisée au moment de l'exécution.
- $T_{réexécution}$ qui est le temps de réexécution du travail perdu. Il correspond à l'exécution des tâches résultant de l'étape de partitionnement statique effectuée précédemment. C'est l'exécution des tâches de communication qui réalise la redistribution des données de manière transparente (que la donnée soit conservée sur un processus de calcul ou sur un serveur de sauvegarde). Il est difficile d'évaluer séparément le temps de redistribution des données et le temps de calcul puisque que l'implémentation de KAAPI été réalisée de manière à ce qu'ils se recouvrent.

Le scénario est le suivant. Durant l'exécution, nous choisissons de réaliser s sauvegardes à intervalle régulier. Si le travail total de l'application correspond à W, alors pour s sauvegardes, l'exécution est partagée en $p+1$ parties de travail $\frac{W}{p+1}$. Nous choisissons également de simuler la défaillance d'une seule machine juste avant la fin de l'exécution ; cela correspond au pire cas pour lequel le travail perdu, et donc le travail à réexécuter pour la reprise, est de $\frac{W}{p+1}$.

Pour réaliser ces mesures, nous nous plaçons sur 110 machines du site de Bordeaux de Grid'5000. 100 machines sont utilisées pour les processus de calcul et 10 machines sont utilisées pour les serveurs de sauvegarde. Nous utilisons la même application de décomposition de domaine que nous décomposons en 1000 sous-domaines, soit 10 sous-domaines par machines. Le domaine de calcul choisi induit un volume total de données à sauvegarder à chaque étape de 4,7 Go.

La figure 8.9 montre le résultat des mesures pour un travail W équivalent en moyenne à environ 210,4 secondes et un nombre de sauvegarde $s \in \{1, 3, 7\}$. Sur la figure, les états ont la signification suivante.

- Calcul correspond au temps d'exécution des calculs de l'application.
- Sauvegarde donne le temps cumulé des s sauvegardes.
- Garde donne le temps de la période de garde, ici 20 secondes pour ces expériences.
- Reprise indique le temps $T_{reprise}$ pour notre protocole de reprise globale dans KAAPI.
- Réexécution du travail perdu correspond au temps $T_{réexécution}$ après reprise pour notre protocole de reprise globale. Ce temps prend aussi en compte la redistribution des données.

Tout d'abord, nous remarquons que dans les trois cas, le temps de reprise $T_{reprise}$ est d'environ 8 secondes. Il semble stable quelque soit la quantité de travail à réexécuter. De plus, ce temps est faible comparé au temps de réexécution du travail perdu qui, dans ce cas, représente la majorité du surcout induit par la panne.

Pour tenter d'évaluer l'influence de la redistribution de données sur le temps de reprise, nous estimons le temps de reprise d'après la vitesse moyenne de calcul des itérations à la reprise. Nous obtenons les valeurs suivantes.

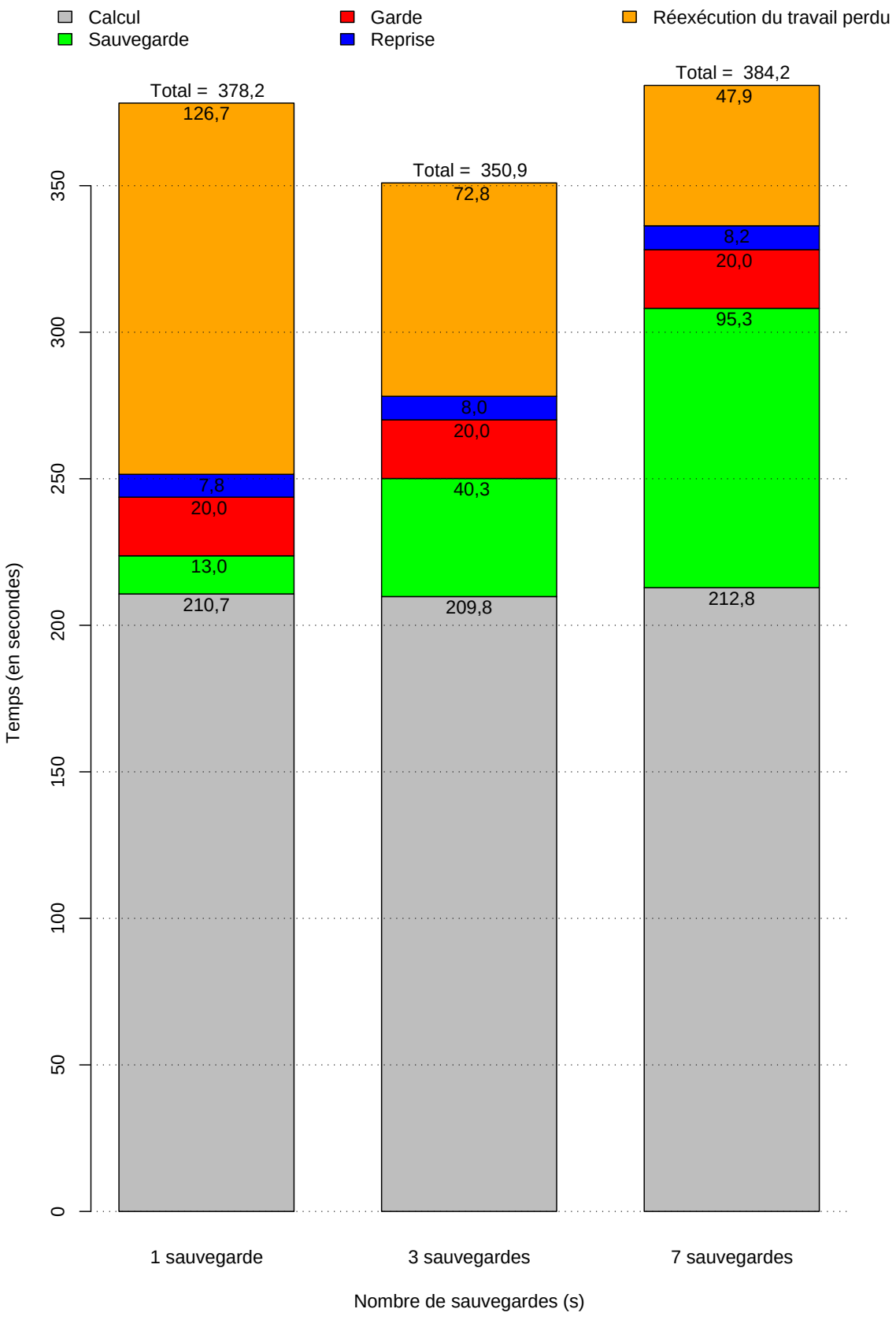


FIG. 8.9: Cout de la reprise globale

	$s = 1$	$s = 3$	$s = 7$
$T_{reprise}$ estimé	117,9 s	59,0 s	29,5 s
$T_{reprise}$ mesuré	126,7 s	72,8 s	47,9 s

Nous constatons une différence de l'ordre de 10 à 20 secondes que nous pouvons imputer à la redistribution des données. Les mesures réalisées ne nous permettent pas d'être plus précis ni de déterminer l'origine exacte de ces différences. Cependant, nous voyons que la redistribution des données est un facteur important puisque, dans le cas de $s = 7$ sauvegardes, elle représente près de 40 % de $T_{reprise}$.

8.6 Conclusion

Dans ce chapitre, nous nous sommes intéressés au protocole de reprise globale associé à la sauvegarde coordonnée. Comme notre l'avons dit dans le chapitre 2, ce protocole est une technique classique de tolérance aux fautes qui est utilisé dans de nombreux environnements.

Nous avons présenté ce protocole de reprise globale et son implémentation dans Kaapi. Puis nous avons modélisé le processus de reprise.

Dans le cas où il n'y a pas de machine de rechange disponible, l'exécution de l'application peut être fortement ralentie après le redémarrage à cause d'un déséquilibrage de charge. Nous avons alors proposé d'utiliser le principe de sur-décomposition pour pallier cet inconvénient. Une modélisation d'une application de décomposition de domaine a été réalisée pour permettre d'estimer la vitesse d'exécution après redémarrage en fonction de la sur-décomposition.

Nous avons également mené plusieurs séries d'expériences afin d'apprécier, sur une application réelle, les gains apportés par la sur-décomposition et également le cout associé à la reprise globale.

Les résultats expérimentaux obtenus confirment les gains attendus sur la vitesse d'exécution au redémarrage sur notre application. Ils mettent également en évidence le surcout lié à la gestion du parallélisme qui est exacerbé lorsque la sur-décomposition devient trop importante. Ainsi, cela permet de mettre en avant le fait que la sur-décomposition ne convient pas aux algorithmiques et aux environnements d'exécution qui ajoutent un surcout important à la parallélisation par rapport à l'exécution séquentielle. Un travail futur sera d'optimiser ce type d'exécution dans le logiciel Kaapi.

Reprise partielle 9

Sommaire

9.1 Introduction

Ce chapitre détaille le protocole de reprise partielle CCK-Restart. C'est un protocole original de reprise partielle qui repose, pour la partie sauvegarde, sur le mécanisme de sauvegarde coordonnée de Kaapi présenté dans le chapitre 7.

L'objectif de ce protocole de reprise partielle est de permettre une reprise plus rapide de l'application en cas de défaillance. Contrairement au protocole de reprise globale vu au chapitre précédent, ce protocole ne réexécute que le travail perdu strictement nécessaire pour redémarrer l'application. En réduisant la quantité de travail à réexécuter, nous réduisons le surcout induit par une défaillance (*cf* la modélisation de la reprise de la section 8.3 page 160).

Comme nous l'avons présenté dans le chapitre précédent et à la différence des approches basées sur la sauvegarde d'un processus au niveau système, nous sommes capables de recharger la représentation abstraite de l'exécution au moment de la sauvegarde. Cette représentation abstraite est un graphe de flot de données sur lequel nous avons appliqué, dans le chapitre précédent, des algorithmes d'ordonnancement afin de rééquilibrer la charge de calcul.

L'idée à la base de cette réduction de la quantité de travail perdu à réexécuter est simple : nous allons profiter de cette représentation abstraite pour ne garder dans ce graphe de flot de données que les tâches **strictement nécessaires** pour permettre de redémarrer l'exécution l'application.

Ce chapitre présente d'une part l'algorithme de calcul du travail à réexécuter strictement nécessaire pour la suite de l'exécution et, d'autre part les résultats de simulations et les résultats expérimentaux que nous avons obtenus sur Grid'5000.

La section suivante présente l'algorithme de reprise partielle qui calcule le travail à réexécuter. La section 9.3.1 estime, à travers des simulations, la quantité de travail à réexécuter pour redémarrer pour une application de type décomposition de domaine. Enfin, les performances de ce protocole de reprise partielle sont évaluées à travers des expérimentations effectuant la reprise partielle d'une application sur une grille de calcul.

9.2 Protocole de reprise partielle

On désire redémarrer l'application après la défaillance d'un ou plusieurs processus. L'application comporte alors deux types de processus : des processus défaillants et des processus non défaillants. Quel que soit le processus P_i, sa dernière sauvegarde G^i est stockée sur un support stable. L'ensemble des sauvegardes de tous les processus constitue un état global cohérent de l'application. De plus, notons que l'état des calculs en cours sur les processus non défaillants est disponible.

9.2.1 Problématique

Dans le cas de la reprise globale, en cas de défaillance d'un ou plusieurs processus, il faut redémarrer tous les processus à partir de leur dernière sauvegarde. Cependant, les calculs réalisés sur tous les processus depuis la dernière sauvegarde sont perdus.

Pour la reprise partielle, on distingue les processus défaillants et les processus non défaillants. Les processus défaillants doivent redémarrer de leur dernière sauvegarde car ils ont perdu tous les calculs effectués depuis leur dernière sauvegarde. Quant aux processus non défaillants, ils conservent leurs calculs en cours. L'état global constitué de l'état sauvegardé des processus défaillants et de l'état courant des processus non défaillants n'est pas cohérent si bien que le calcul ne peut pas continuer à partir de cet état.

L'objectif du protocole de reprise partielle est de rendre cohérent cet état global de manière à pouvoir reprendre le calcul en ayant perdu un minimum de travail. Pour cela, on demande aux processus non défaillants de réexécuter des tâches supplémentaires (extraites de leur dernière sauvegarde) afin de se ramener à un état global cohérent. Pour déterminer cet ensemble des tâches à réexécuter, il faut d'abord identifier l'ensemble des communications à destination des processus défaillants qui ont été perdues. L'ensemble

des tâches à réexécuter est alors constitué des tâches nécessaires pour émettre à nouveau ces communications. Cette technique nous permet de réduire le nombre de tâches à réexécuter en permettant aux processus non défaillants de conserver le bénéfice des calculs déjà effectués.

La suite de cette section décrit la méthode utilisée par ce protocole de reprise partielle pour calculer l'ensemble des communications à rejouer et des calculs à refaire sur chaque processus de manière à garantir un état global cohérent lorsque seuls les processus défaillants sont redémarrés à partir de leur dernière sauvegarde.

9.2.2 Calcul du travail à réexécuter

Nous proposons un algorithme qui permet de calculer le travail à réexécuter nécessaire pour redémarrer une application par reprise partielle. Cette méthode travaille sur le graphe de flot de données de la dernière sauvegarde, noté G^i. Pour les processus non défaillants, les informations concernant l'état d'exécution courant sont ajoutées à ce graphe. Les sommets tâches de ce graphe G^i sont annotés en fonction de leur état : exécutés ou non exécutés.

Cette méthode est distribuée et une difficulté réside dans le fait que rejouer une communication perdue peut entrainer également la nécessité de rejouer d'autres communications, en particulier sur des processus qui ne communiquent pas directement avec les processus défaillants.

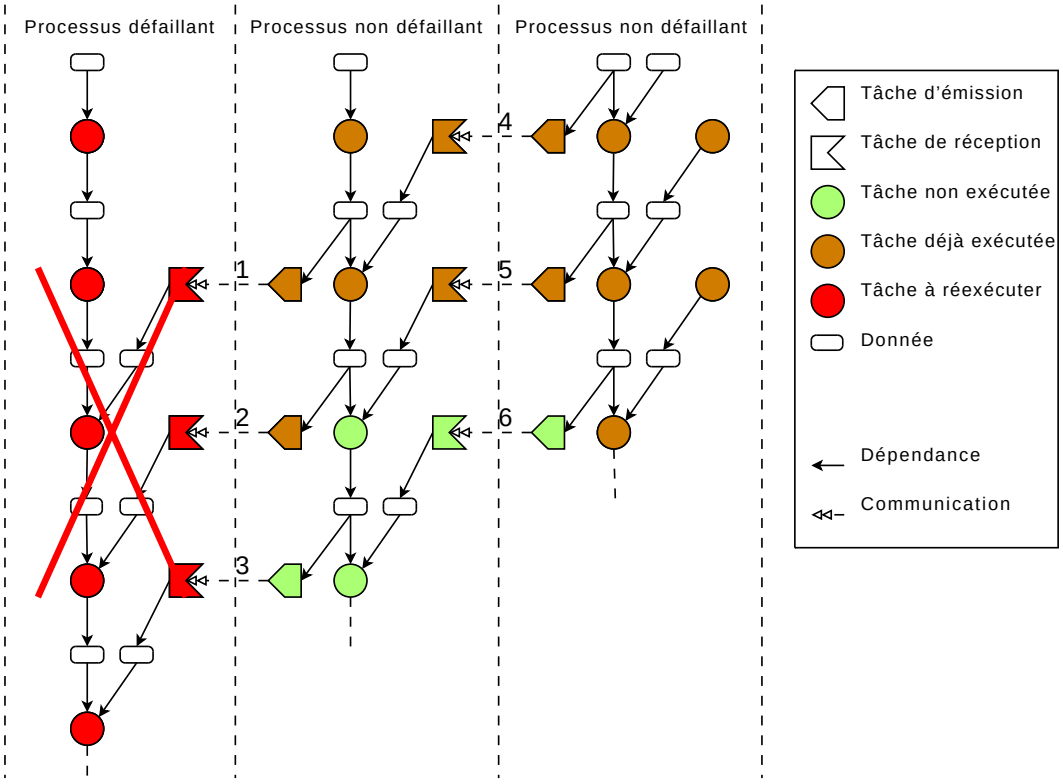


FIG. 9.1: Exemple avec un processus défaillant et deux processus non défaillants ; les tâches déjà exécutées sont marquées.

La figure 9.1 montre un exemple de l'état d'une application au début de la reprise avec un processus défaillant et deux processus non défaillants. Les tâches qui ont déjà été exécutées sont marquées ; le processus défaillant, à gauche, a bien entendu perdu toutes les tâches qu'il avait exécutées.

Les sections suivantes définissent les graphes et les ensembles qui sont utilisés dans l'algorithme présenté à la section 9.2.3.

9.2.2.1 Notations

Par la suite, nous utilisons la notation X^i pour faire référence à un graphe ou à un ensemble associé au processus P_i, tandis que X fait référence à un graphe ou à un ensemble global à tous les processus. X correspond à la réunion des X^i de tous les processus P_i.

Soit $G = (S, A)$ un graphe avec S l'ensemble des sommets et A l'ensemble des arcs. On note $G^* = (S, A^*)$ la fermeture transitive de ce graphe.

Graphe de flot de données. La sauvegarde d'un processus P_i est constitué du graphe de flot de données G^i et des versions des données en entrée.

Un graphe de flot de données est un graphe orienté acyclique $G = (S, A)$. C'est aussi un graphe biparti entre les tâches ($\in S_T$) et les versions ($\in S_V$) (on a $G = S_T \cup S_V$). Chaque sommet tâche est connecté à un ou plusieurs sommets versions et chaque sommet version est connecté à un ou plusieurs sommets tâches comme présenté à la section 4.2.3.

Tâches de communication. Parmi les sommets tâches, nous distinguons les tâches de communication (*cf* section 4.3.2.3 page 87). Une tâche de communication peut être soit une émission (`Broadcast`), soit une réception (`Receive`)[1]. On notera C le sous-ensemble de S_T des sommets tâches qui sont des tâches de communication.

À chaque émission est associée une unique réception et inversement. Ce couple est appelé une communication et est identifié par un identifiant unique. Nous soulignons de plus que les communications n'interviennent qu'entre deux processus distincts et donc qu'on ne peut avoir l'émission et la réception d'une même communication dans le même graphe G^i d'un processus P_i.

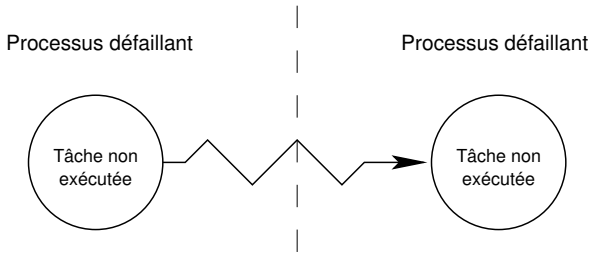
Nous désignons une communication par son identifiant c. Soit c une communication, $émission(c) \in S_T$ est la tâche d'émission associée à la communication c et $réception(c) \in S_T$ est la tâche de réception associée à la communication c.

9.2.2.2 Communications perdues

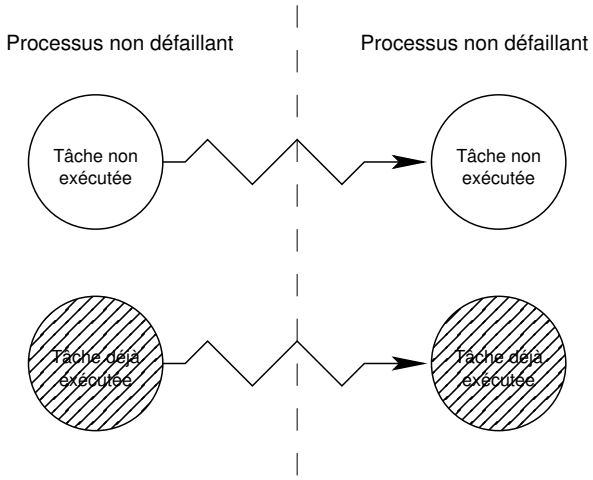
Nous cherchons tout d'abord à déterminer les communications qui doivent être rejouées pour rétablir l'application dans un état cohérent. Plusieurs cas de figure peuvent se présenter selon l'état des processus qui communiquent.

[1]Pour simplifier, nous ignorons ici les tâches `Wait` (*cf* section 4.3.2.3 page 4.3.2.3).

Communication entre deux processus défaillants. Les deux processus redémarrent à partir de leur dernière sauvegarde, toutes les tâches de communication entre les deux processus seront rejouées lors de la reprise. Il n'y a donc rien à rejouer.

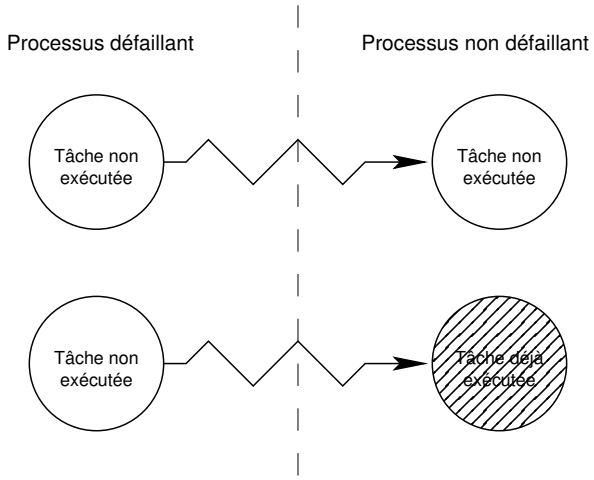


Communication entre deux processus non défaillants. Les deux processus ont exécuté des tâches de communication. Cependant, l'état de ces processus résulte d'une exécution normale, les communications entre ces deux processus sont dans un état cohérent. Il n'y a rien à rejouer.



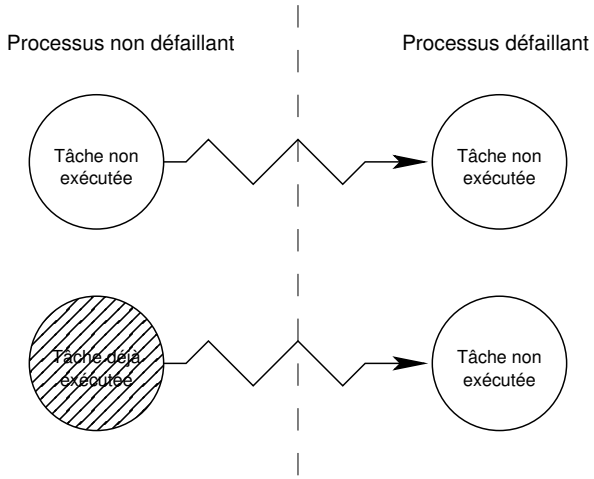
Communication d'un processus défaillant vers un processus non défaillant. Le processus défaillant redémarre à partir de sa dernière sauvegarde et il va donc réémettre toutes les communications de son graphe. Pour le processus non défaillant, il faut distinguer deux cas selon l'état de la réception correspondante :

- Si la réception n'a pas été effectuée, ceci est cohérent avec l'état de l'émission. Il n'y a rien à rejouer.
- Si la réception a déjà été effectuée, l'état est incohérent. Cependant, cette communication n'est pas utile au processus non défaillant pour continuer son calcul. Dans ce cas, le processus défaillant doit simplement ne pas réémettre la communication.



Communication d'un processus non défaillant vers un processus défaillant. Le processus défaillant redémarre à partir de sa dernière sauvegarde et donc aucune de ses réceptions n'a été effectuée. Il a donc besoin de toutes les émissions correspondantes en provenance du processus non défaillant pour reprendre son exécution. Ainsi, pour le processus défaillant, cela dépend de l'état de l'émission correspondante :

- Si l'émission n'a pas été effectuée, on a un état cohérent et il n'y a rien à rejouer.
- Si l'émission a déjà été effectuée, on a besoin de la rejouer et aussi de réexécuter toutes les tâches nécessaires à la production de la donnée communiquée.



Pour résumer, rejouer une tâche de communication n'est nécessaire que dans un seul cas : quand cette communication se fait d'un processus non défaillant à un processus défaillant et que cette communication a déjà été effectuée depuis la dernière sauvegarde.

Définition 13 *L'ensemble des* ***communications perdues*** $C_{perdues}$ *est l'ensemble des tâches d'émission d'un processus non défaillant vers un processus défaillant qui ont déjà été exécutées depuis la dernière sauvegarde.*

Cet ensemble correspond à l'ensemble des messages qui ne seront pas réémis par l'état courant des processus non défaillants et l'état sauvegardé des processus défaillants.

Ces communications correspondent aux évènements non déterministes qui ne peuvent être réémis par l'état courant. Forcer la réémission de ces communications permet donc d'éviter les processus orphelins.

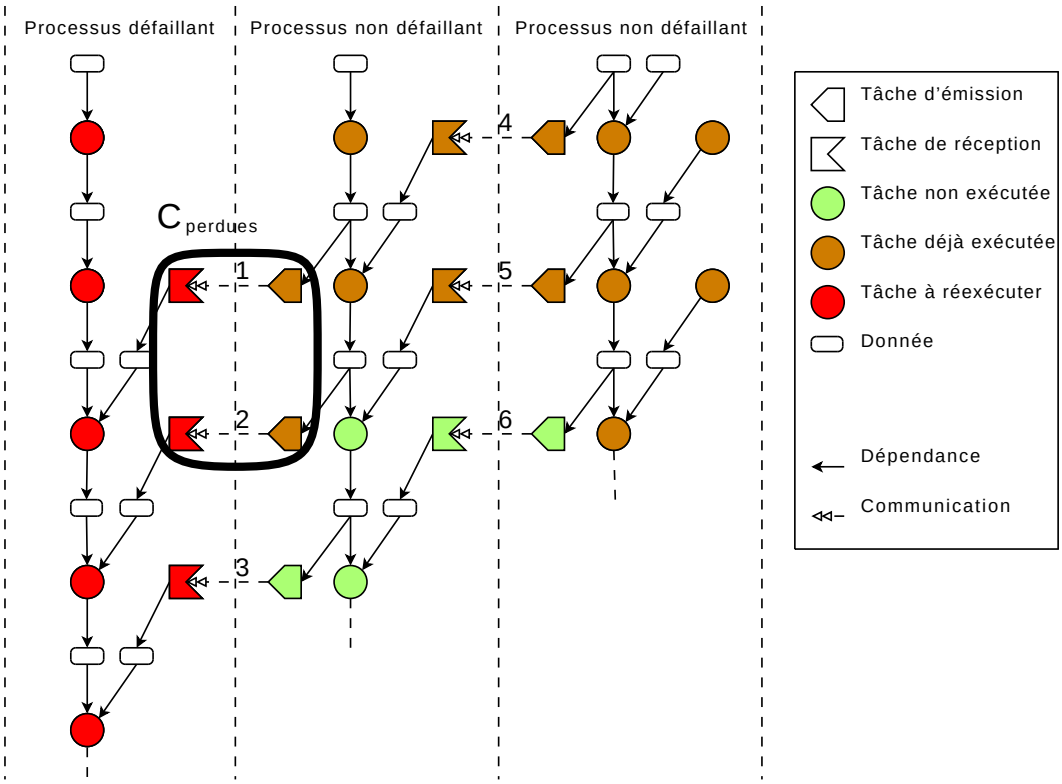


FIG. 9.2: L'ensemble des communications perdues

Sur la figure 9.2, on peut voir l'ensemble des communications perdues pour notre exemple ; ce sont les communications numérotées 1 et 2.

9.2.2.3 Graphe restreint aux communications

Nous définissons le graphe restreint aux communications $\overline{G^i}$ qui représente les dépendances entre les émissions et les réceptions de données au sein d'un processus P_i. Ce graphe permet de calculer l'ensemble des données à recevoir pour calculer de manière globale l'ensemble des tâches à réexécuter pour réémettre les communications perdues vers le processus défaillant.

Définition 14 *Le* ***graphe restreint aux communications*** $\overline{G^i} = (\overline{S^i}, \overline{A^i})$ *est défini tel que :*

- *Les sommets* $\overline{S^i}$ *sont les tâches de* S^i *qui sont des communications* ($\overline{S^i} = C^i$).
- *Les arcs* $\overline{A^i}$ *sont définis de la manière suivante :* $\forall r, e \in \overline{S^i}$, $(r, e) \in \overline{A^i}$ *si et seulement si*
  - r *est une réception,*
  - e *est une émission,*
  - *et s'il existe un chemin de* e *à* r.

Une définition équivalente de $\overline{G^i}$ est que le graphe restreint aux communications $\overline{G^i}$ est le sous-graphe induit par les sommets des tâches de communication de la fermeture transitive de G^i. Ainsi, pour $r, e \in \overline{S^i}$, l'émission de e par P_i nécessite d'abord la réception de r si et seulement s'il existe dans $\overline{G^i}$ un arc allant de r à e.

De même, nous définissons le graphe restreint global $\overline{G} = \bigcup_i \overline{G^i}$ qui représente les dépendances entre les communications entre tous les processus.

9.2.2.4 Ensemble des communications à rejouer

L'ensemble $C_{perdues}$ des communications perdues contient les communications à rejouer pour rétablir l'application dans un état cohérent. Cependant, pour rejouer ces tâches, il est nécessaire de réexécuter les tâches de calcul qui permettent de produire les données communiquées. Ces tâches de calcul peuvent aussi nécessiter la réception de données. Le graphe global restreint aux communications $\overline{G} = \bigcup_i \overline{G^i}$ permet, grâce aux dépendances entre les communications qu'il contient, de déterminer l'ensemble total $C_{totales}$ des communications à rejouer pour réémettre toutes les communications perdues.

Définition 15 *L'ensemble des* ***communications à rejouer*** $C_{totales}$ *est l'ensemble des communications du graphe* $\overline{G}$ *qui précédent les tâches appartenant à* $C_{perdues}$*.*

Formellement, si $\overline{G}^* = (\overline{S}, \overline{A}^*)$ est la fermeture transitive de $\overline{G} = \bigcup_i \overline{G^i}$, on a

$$C_{totales} = C_{perdues} \cup \bigcup_{c \in C_{perdues}} \{c' \text{ tels que } (\mathit{réception}(c'), \mathit{émission}(c)) \in \overline{A}^*\}$$

$C^i_{totales}$ est alors défini comme l'ensemble des communications à rejouer appartenant au processus P_i.

Sur la figure 9.3, l'ensemble total des communications à rejouer correspond aux communications 1, 2 et 4. La communication 4 doit être rejouée car la communication 2 dépend de 4.

9.2.2.5 Ensemble des tâches à réexécuter

Une fois que chaque processus P_i connait son ensemble $C^i_{totales}$, il peut déterminer l'ensemble $G^i_{réexécution}$ des tâches de calcul à réexécuter. Ce sont les tâches dont dépendent les tâches d'émission des communications de $C^i_{totales}$. L'ensemble $G^i_{réexécution}$ contient l'intégralité des tâches à réexécuter pour le processus P_i. En effet, si une de ces tâches de calcul nécessite une communication, elle a été prise en compte dans l'ensemble $C^j_{totales}$ du processus voisin P_j grâce au calcul de la fermeture transitive G^*.

Si $G^* = (S, A^*)$ est la fermeture transitive de G, on a

$$G^i_{réexécution} = \mathit{émission}(C_{totales}) \cup \bigcup_{c \in C^i_{totales}} \{t \in S^i \text{ tels que } (t, \mathit{émission}(c)) \in A^*\}$$

Sur la figure 9.4, on peut voir l'ensemble des tâches à réexécuter pour reprendre le calcul. Ce sont toutes les tâches nécessaires pour rejouer les communications de $C_{totales}$. $G_{défaillant}$ est le graphe des processus défaillants ; ses tâches doivent également être réexécutées.

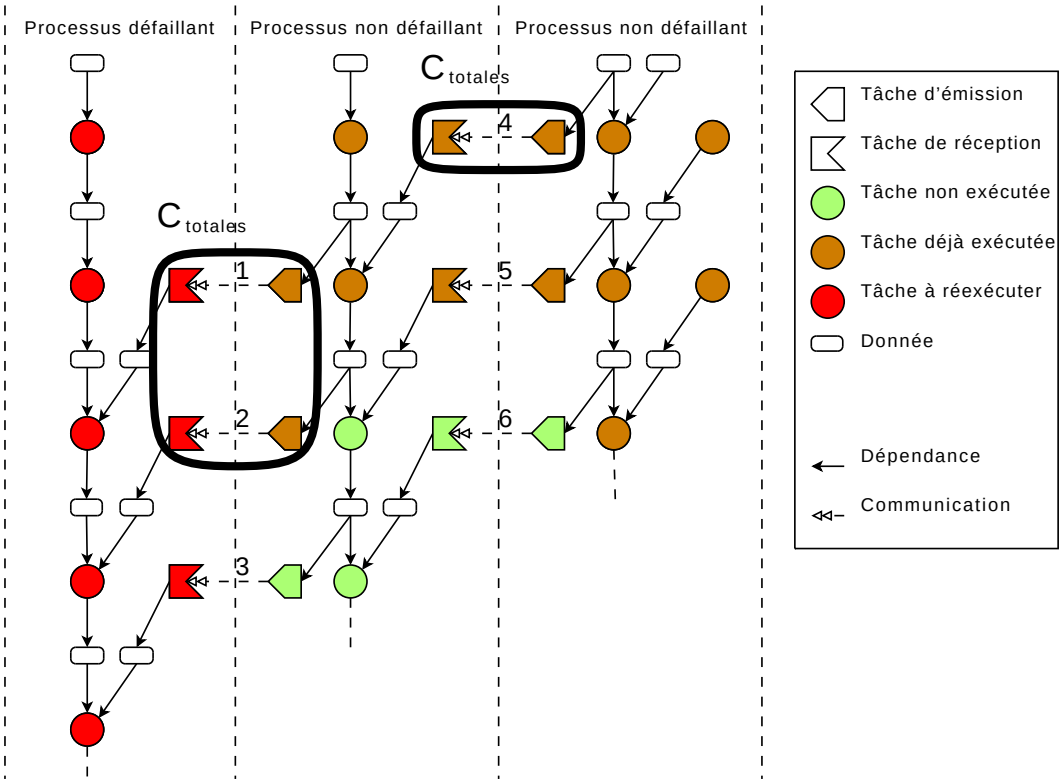


FIG. 9.3: L'ensemble total des communications à rejouer

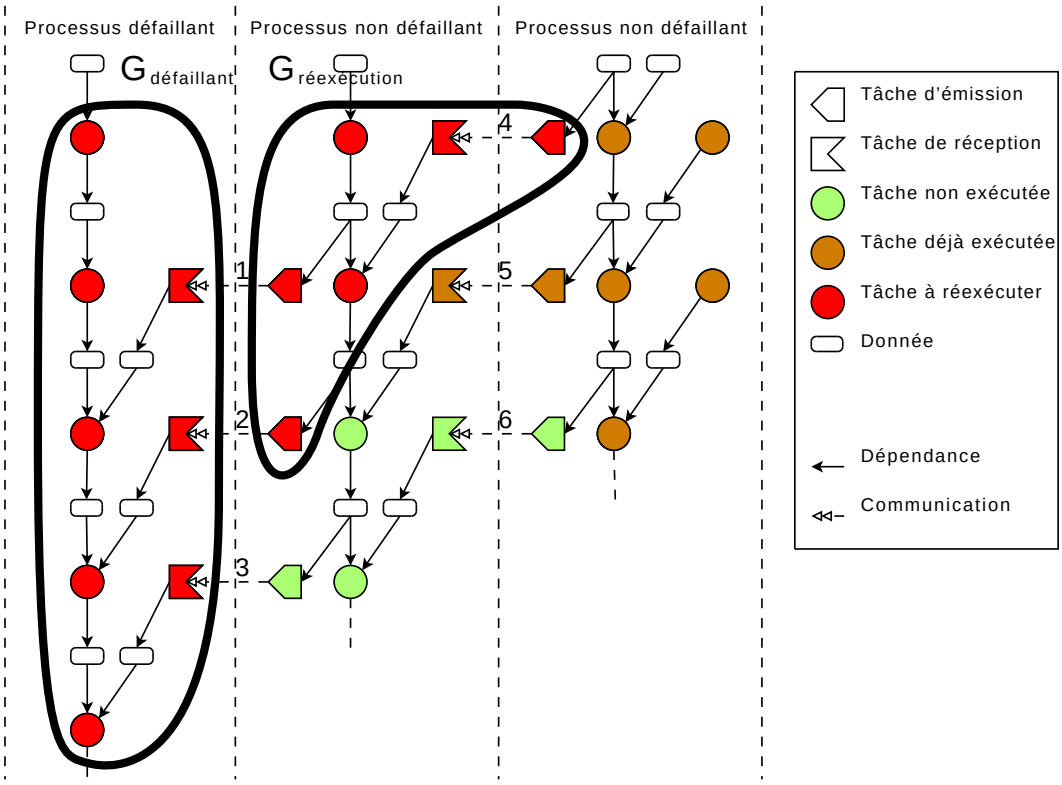


FIG. 9.4: L'ensemble des tâches à réexécuter

9.2.3 Algorithme

En nous basant sur la construction de $G_{réexécution}$ donnée précédemment, nous proposons un algorithme distribué qui permet le calcul des tâches à réexécuter pour effectuer une reprise partielle. Cet algorithme s'exécute sur chaque processus, défaillant ou non, lors d'un redémarrage. Dans le cas où il n'y a pas de processus de rechange pour remplacer les processus défaillants, le redémarrage de chaque processus défaillant est pris en charge par un processus non défaillant. Ce dernier exécute alors l'algorithme à la fois pour son propre état et aussi pour l'état des processus défaillants qu'il remplace.

Cet algorithme s'exécute localement sur chaque processus en utilisant le mécanisme de reconfiguration présenté dans le chapitre 5. Cette reconfiguration « reprise partielle » s'applique sur tous les processus non défaillants en cohérence mutuelle. Elle travaille sur le graphe de la dernière sauvegarde sur lequel sont ajoutées les informations concernant l'état d'exécution courant.

Les processus qui remplacent les processus défaillants sont désignés de manière centralisée durant le prologue de la reconfiguration. Ce sont soit de nouveaux processus lancés sur des machines de rechange, soit des processus existants qui effectuent la reprise des processus défaillants en plus de la leur.

La fonction de reconfiguration « reprise partielle » qui est exécutée sur chacun des processus suit l'algorithme suivant.

1. Construction du graphe restreint aux communications $\overline{G^i}$
2. Diffusion de $\overline{G^i}$ et de l'état de chaque communication à tous les processus
3. Calcul de l'ensemble $C_{perdues}$ des communications perdues
4. Calcul de l'ensemble $C^i_{totales}$ des communications à rejouer
5. Calcul de l'ensemble $G^i_{réexécution}$ des tâches à réexécuter
6. Reconstruction de l'état du processus à partir de $G^i_{réexécution}$: les tâches à réexécuter sont conservées dans l'état du processus, les autres sont supprimées.

Le protocole de reprise partielle présenté dans ce chapitre est basé sur la technique de sauvegarde coordonnée. Cependant, son principe est proche des protocoles de tolérance aux fautes par journalisation.

En effet, pour les protocoles de reprise par journalisation, seuls les processus défaillants repartent à partir de la dernière sauvegarde. Le rôle du protocole de reprise par journalisation est alors de rejouer les évènements non déterministes à destination des processus qui sont retournés en arrière de manière à retrouver un état identique à celui d'avant la panne. Habituellement, ces évènements non déterministes ont été sauvegardés sur la mémoire stable ce qui permet de les rejouer facilement.

Dans le cas de notre protocole de reprise partielle, le fonctionnement est identique à la seule différence que les évènements non déterministes (dans notre cas ce sont les communications) ne sont pas sauvegardés. Nous proposons à la place de réémettre ces communications en réexécutant les calculs qui ont permis de générer ces communications.

L'utilisation d'une sauvegarde coordonnée qui garantit un état global sauvegardé cohérent permet de nous assurer que tous les messages qui ont été émis depuis la dernière sauvegarde peuvent être recalculés. Le pire cas correspond alors au cas où il

est nécessaire de réexécuter tous les calculs depuis la dernière sauvegarde, ce qui est alors équivalent à une reprise globale.

Les protocoles par journalisation classiques présentent un surcout lors de l'exécution sans panne, notamment du point de vue de l'utilisation du réseau. Notre protocole reporte ce surcout uniquement au moment de la reprise en termes de calcul nécessaire pour regénérer les messages non sauvegardés.

9.2.3.1 Cohérence de l'état reconstruit

Cette partie montre que l'état global reconstruit à la reprise est un état global cohérent. Pour cela, nous rappelons les points suivants :

- L'état constitué de l'ensemble des dernières sauvegardes est un état global mutuellement cohérent car il repose sur la technique de sauvegarde coordonnée présentée à la section 7.3. En particulier, l'état des canaux de communication est nul.
- Durant la reprise, et donc pendant l'exécution de cet algorithme, les calculs sont arrêtés et les processus sont dans un état mutuellement cohérent : les graphes G^i sont donc figés et les canaux de communication sont vides.

Définition 16 *On définit l'**état global reconstruit** comme l'état de l'application à la fin de la reprise.*

L'état global reconstruit est constitué de la réunion des éléments suivants.

- *L'état $G_{exécution}$ des processus non défaillants en cours d'exécution au début de la reprise (l'état des processus défaillants est nul).*
- *L'état $G_{défaillant}$ des processus défaillants restaurés à partir de la sauvegarde.*
- *L'ensemble $G_{réexécution}$ des tâches à réexécuter.*

De plus, les canaux de communication sont vides. Leur état n'intervient donc pas dans l'état global reconstruit.

Proposition 6 *L'état global reconstruit est un état global cohérent.*

Preuve Un état global non cohérent est caractérisé par la présence de processus orphelins. Nous montrons que l'état global reconstruit ne comporte pas de processus orphelins, c'est-à-dire que tous les messages dont dépendent les processus peuvent être réémis. En termes de tâches pour notre modèle, cela signifie qu'il ne doit pas y avoir de communication qui a une tâche de réception sans tâche d'émission associée.

Par définition, l'ensemble des communications perdues correspond à l'ensemble des communications du graphe $G_{exécution} \cup G_{défaillant}$ ne pouvant être réémises.

Nous allons montrer que :

1. les tâches d'émission associées aux communications perdues $C_{perdues}$ sont dans $G_{réexécution}$;
2. l'ensemble $G_{réexécution}$ ne comporte pas de réception sans émission.

Tout d'abord, on a bien $émission(C_{perdues}) \subset G_{réexécution}$ car d'après les définitions de $C_{totales}$ et $G_{réexécution}$, on a $émission(C_{perdues}) \subset C_{totales}$ et $émission(C_{totales}) \subset G_{réexécution}$. Ceci montre le point 1.

Ensuite, l'ensemble des communications de $G_{réexécution}$ correspond à l'ensemble $C_{totales}$ qui est un sous-ensemble de l'ensemble des communications de G. G est l'état global de la dernière sauvegarde et cet état est mutuellement cohérent, donc pour chaque tâche de réception, la tâche d'émission associée existe également. De plus, d'après la définition de $C_{totales}$, si une communication $c \in C_{totales}$ alors $émission(c) \in G_{réexécution}$. Ceci montre le point 2.

Tout ceci nous garantit que l'état global reconstruit est un état global cohérent. □

9.2.3.2 Analyse de cout

Nous analysons le cout de l'algorithme de reprise partielle proposé, en temps et en nombre de messages.

Graphe restreint aux communications. Le graphe G^i est un graphe orienté acyclique. Le calcul de la fermeture transitive peut donc se faire en $O(|S^i| + |A^i|)$ [55]. Il en est de même pour l'extraction du sous-graphe. De plus, ces calculs sont faits en parallèle, chaque processus P_i calcule son graphe $\overline{G^i}$ à partir de G^i. Le cout en temps de calcul du graphe restreint est donc en $O(max_i(|S^i| + |A^i|))$.

Diffusion des graphes restreints. Cette étape effectue la diffusion de tous les graphes $\overline{G^i}$ à tous les processus. Cette communication collective est un schéma du type *Gather-to-All*. Ce type de communication peut être réalisé avec un cout en temps de $O(log_2\, n)$ messages [116, 144] (avec n le nombre de processus participant).

Ensemble des communications perdues. Pour déterminer l'ensemble $C_{perdues}$ des communications perdues, il suffit de parcourir le graphe global $\overline{G}$ et de regarder l'état des tâches de communication correspondantes (émissions et réceptions).

Le cout en temps de ce calcul est donc en $O(|\overline{S}|)$.

Ensemble total des communications à rejouer. La détermination de cet ensemble $C_{totales}$ peut se faire par le calcul de la fermeture transitive du graphe $\overline{G}$. Ce graphe est un graphe orienté acyclique. Ce calcul peut se faire en temps $O(|\overline{S}| + |\overline{A}|)$ [55].

Ensemble des tâches à réexécuter. L'ensemble $G^i_{réexécution}$ des tâches à réexécuter sur le graphe G^i sont les tâches dont dépendent les communications de $C_{totales}$. La détermination de cet ensemble peut encore se faire par le calcul de la fermeture transitive du graphe G^i orienté et acyclique, qui a déjà été effectué pour le calcul du graphe restreint.

Gestion de la cohérence mutuelle. Cet algorithme est exécuté comme une reconfiguration nécessitant la cohérence mutuelle. La complexité de la gestion de la cohérence mutuelle pour Kaapi est en $O(nv)$, avec n le nombre de processus participant et v le nombre moyen de voisins (*cf* section 5.4.3.3).

Cout total. Le cout en temps de cet algorithme permettant le calcul de l'ensemble des tâches à réexécuter est donc en $O(|\overline{S}| + |\overline{A}|)$. Le cout en nombre de messages pour cet algorithme est en $O(nv)$.

9.2.4 Amélioration

Nous proposons également une amélioration qui permet de réduire le nombre de tâches à réexécuter. Cette optimisation consiste à tenir compte des versions de données présentes en mémoire sur les processus non défaillants.

Dans le modèle d'exécution Kaapi, les tâches exécutées sont détruites au fur et à mesure. Pourtant, il est possible qu'une version d'une donnée soit toujours disponible en mémoire si elle doit être lue par une tâche qui n'a pas encore été exécutée. Si une telle donnée est dans le graphe des tâches à réexécuter, elle peut être utilisée pour éliminer des tâches de calcul de l'ensemble des tâches à réexécuter.

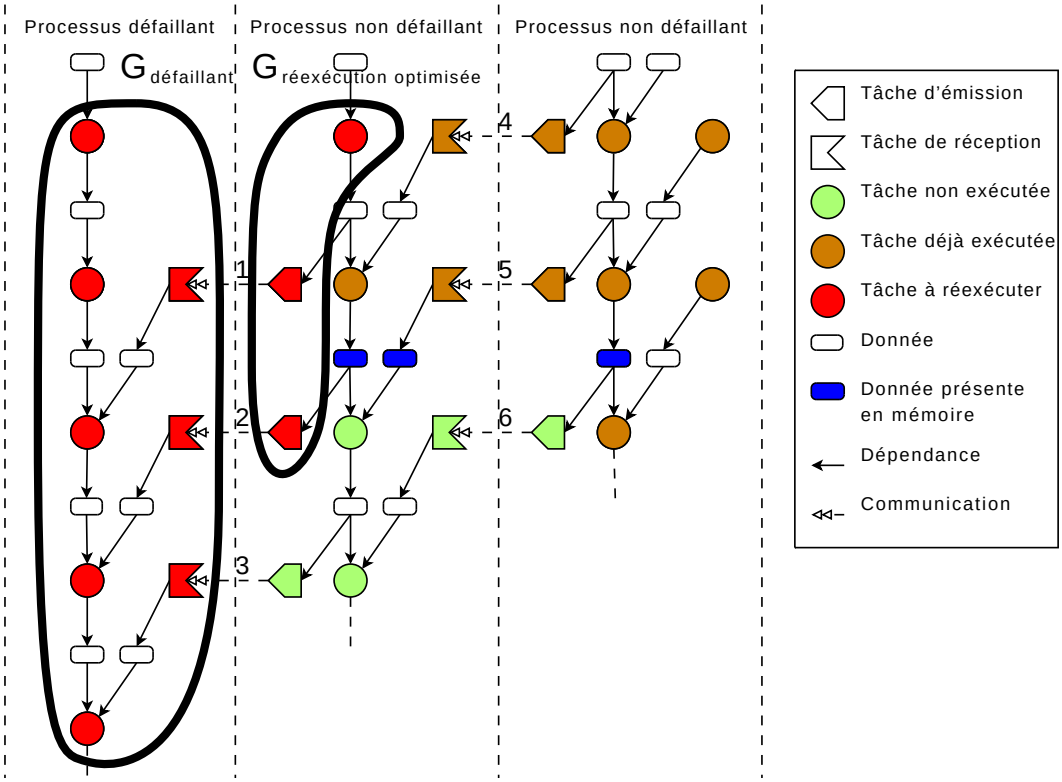


Fig. 9.5: L'ensemble des tâches à réexécuter en prenant en compte les données disponibles en mémoire

Sur la figure 9.5, nous pouvons voir l'ensemble des tâches à réexécuter pour reprendre le calcul en prenant en compte des données présentes en mémoire. En effet, rejouer la communication 2 ne nécessite pas de réexécuter d'autres tâches puisque la version de la donnée est toujours présente en mémoire. Ceci permet également d'éviter de rejouer la communication 4 et de propager la reprise partielle au troisième processus.

9.3 Réexécution du travail perdu

Nous reprenons la modélisation de la reprise de la section 8.3 pour l'adapter au cas de notre protocole de reprise partielle. Le surcout induit par une défaillance s'exprime de la manière suivante :

$$T_{panne} = T_{garde} + T_{reprise} + T_{réexécution}$$

- T_{garde} est indépendant de la méthode reprise donc il ne change pas.
- $T_{reprise}$ correspond au temps d'exécution de l'algorithme de reprise partielle qui calcule l'ensemble des tâches à réexécuter pour la reprise. Cet algorithme a été présenté à la section 9.2.3 et son cout a été évalué à la section 9.2.3.2. Les expériences de la section 9.4 montreront que ce temps est généralement faible comparé à $T_{réexécution}$.
- $T_{réexécution}$ est le temps nécessaire pour réexécuter le travail perdu W_{perdu}. La technique de reprise partielle présentée dans ce chapitre permet de réduire cette quantité de travail par rapport au cas de la reprise globale. Grâce à un bon ordonnancement des tâches, il est alors possible de réduire ce temps de réexécution du travail perdu.

La différence entre les deux protocoles concerne la définition du travail perdu. Nous notons respectivement $W_{perdu}^{globale}$ et $W_{perdu}^{partielle}$ le travail perdu pour la reprise globale et la reprise partielle. De même, $T_{réexécution}^{globale}$ et $T_{réexécution}^{partielle}$ désignent respectivement le temps de réexécution du travail perdu dans le cas de la reprise globale et de la reprise partielle.

Travail perdu. Nous considérons une exécution sur n machines avec la défaillance de p machines. Nous simplifions de plus le problème en considérant que le travail est équilibré entre les machines et que le redémarrage se fait sur n machines (grâce à des machines de remplacement).

- $W_{perdu}^{globale}$ est l'ensemble des instructions exécutées depuis la dernière sauvegarde.
- $W_{perdu}^{globale}/n$ est le travail exécuté par chaque processus depuis la dernière sauvegarde.
- $p \times W_{perdu}^{globale}/n$ est la partie du travail perdu due aux processus défaillants.
- $\varepsilon \times W_{perdu}^{globale}/n$ est la partie du travail perdu due aux processus voisins des processus défaillants qui doivent rejouer des communications.

Nous exprimons donc le travail perdu de la reprise partielle en fonction ce celui de la reprise globale :

$$W_{perdu}^{partielle} = \frac{p+\varepsilon}{n} \times W_{perdu}^{globale}$$

ε est la proportion (par rapport à $n-p$) de tâches à réexécuter sur les processus non défaillants. Elle est difficile à évaluer puisqu'elle dépend de très nombreux paramètres. Elle dépend, entre autres, de l'application et du nombre de processeurs sur lesquels elle s'exécute. En particulier, les motifs de communication entre les processus sont déterminants ; ce sont ces communications qui créent des dépendances entre les processus et imposent la réémission de certains messages. Cette valeur dépend également de la date de la panne et de la date la dernière sauvegarde. Dans tous les cas, cette valeur reste

bornée par $n - p$: le pire cas étant le cas où toutes les tâches doivent être réexécutées ce qui correspond à la reprise globale.

La section suivante présente des simulations qui ont pour but d'évaluer le comportement du rapport $W_{perdu}^{globale}/W_{perdu}^{partielle}$ en fonction de différents paramètres.

Temps de réexécution. Le protocole de reprise partielle présenté ici permet d'avoir une quantité de travail perdu plus faible qu'avec le protocole de reprise globale. Cela permet d'avoir une quantité de travail à réexécuter plus faible à la reprise. Si le travail a été initialement sur-décomposé, il est alors possible de paralléliser cette quantité de travail afin de réduire le surcout induit par la panne T_{panne}.

Si le travail perdu $W_{perdu}^{partielle}$ peut être réparti de manière équilibrée sur tous les processus, on a alors :

$$T_{réexécution}^{partielle} = \frac{p + \varepsilon}{n} \times T_{réexécution}^{globale}$$

La figure 9.6 compare le travail perdu et le temps de réexécution dans le cas de la reprise globale et dans le cas de la reprise partielle. Dans les deux cas, le processus P_3 tombe en panne et une machine de rechange est disponible pour relancer le processus P_3.

Pour la reprise globale sur la sous-figure 9.6a en haut, le travail perdu $W_{perdu}^{globale}$ est constitué de tout le travail qui a été exécuté entre la dernière sauvegarde et la défaillance.

Sur la sous-figure 9.6b en bas, la reprise partielle permet de réduire la quantité de travail perdu $W_{perdu}^{partielle}$. La redistribution de ce travail sur toutes les machines permet d'obtenir un temps de réexécution $T_{réexécution}^{partielle}$ plus court que dans le cas de la reprise globale. Bien sûr, cette redistribution du travail a un cout direct : celui de redistribuer les données associées. Nous étudierons cela de manière expérimentale dans la suite de ce chapitre.

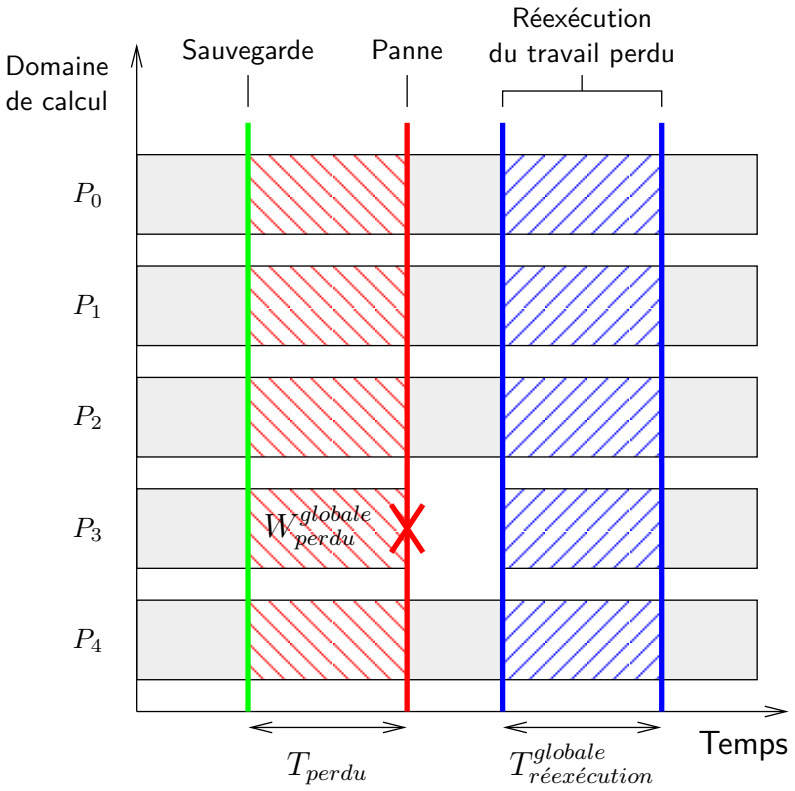
9.3.1 Simulations

Dans le but d'évaluer la quantité de travail à réexécuter pour le protocole de reprise partielle, nous avons réalisé des simulations de redémarrage. Les courbes suivantes sont le résultat de simulations de l'étape de redémarrage après défaillance d'un seul processus.

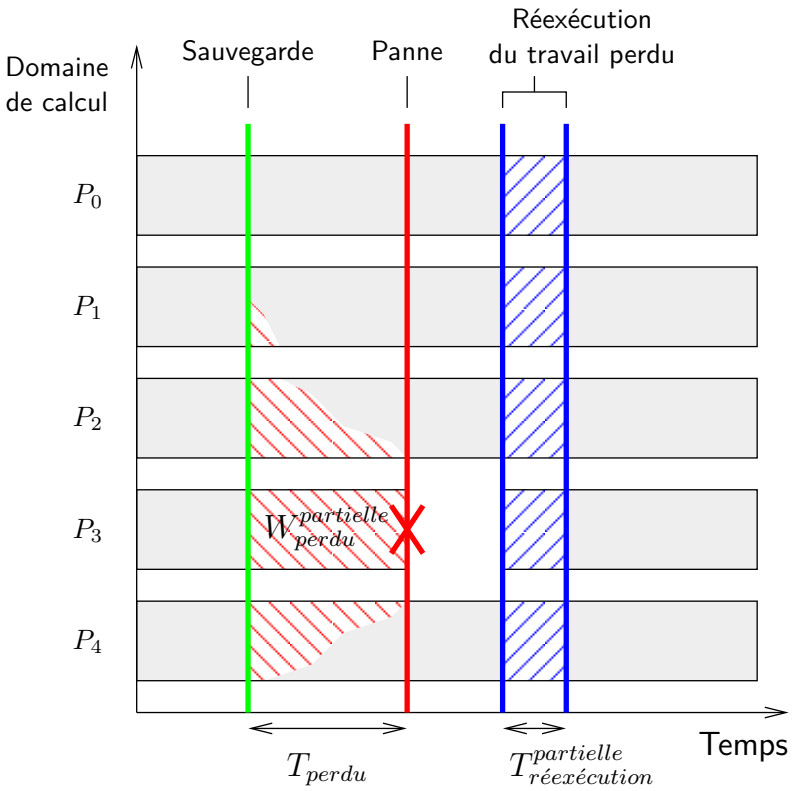
9.3.1.1 Scénario

Pour les besoins de ces simulations, nous considérons la même application que dans le chapitre précédent, à savoir une résolution du problème de Poisson par la méthode itérative de Jacobi sur un domaine à trois dimensions. Nos simulations suivent la modélisation d'application donnée dans la section 8.4.1 page 163.

Le domaine utilisé est de taille $2\ 048^3$ (soit 64 Go de données) découpé en 64^3 sous-domaines de 32 Ko chacun. À chaque itération du calcul, la mise à jour d'un sous-domaine correspond à une tâche de calcul. Le calcul de cette tâche nécessite la connaissance des 6 sous-domaines voisins (un voisin pour chaque face du cube représentant le sous-domaine en 3 dimensions). Sur la machine de référence (Bi-Opteron



(a) Reprise globale



(b) Reprise partielle

FIG. 9.6: Réexécution du travail perdu après la défaillance du processus P_3

à 2 Ghz avec 2 Go de mémoire RAM), une tâche de calcul d'un sous-domaine s'exécute en 10 ms.

Grâce à ces simulations nous étudions deux valeurs.

- Les tâches à réexécuter désignent l'ensemble des tâches de mise à jour d'un sous-domaine qui doivent être réexécutées pour redémarrer correctement l'application. Ce nombre de tâches nous permet d'évaluer la quantité de travail perdu $W_{perdu}^{partielle}$. Cette valeur est tracée proportionnellement au nombre de tâches à réexécuter dans le cas de la reprise globale ; ce qui correspond donc au rapport $W_{perdu}^{partielle}/W_{perdu}^{globale}$.
- Les processus impliqués désignent les processus qui contiennent au moins une tâche de calcul à réexécuter avec la reprise partielle. Les autres processus, ceux qui ne sont pas impliqués, sont ceux qui n'ont pas de travail perdu à cause de la défaillance. Cependant, ces processus peuvent tout de même être utilisés lors de l'équilibrage du travail à réexécuter.

Dans nos simulations, les sauvegardes sont réalisées périodiquement et la quantité de travail à réexécuter calculée est celle du pire cas, c'est-à-dire quand la défaillance se produit juste avant la prochaine sauvegarde. Une autre manière d'interpréter la période de sauvegarde est de dire qu'elle désigne la durée entre la dernière sauvegarde et la défaillance puisque nous nous plaçons en pire cas.

9.3.1.2 Influence de la période de sauvegarde

Pour cette simulation, nous considérons que l'application est distribuée sur 1 024 processeurs et nous associons un processus à chaque processeur. Les 64^3 sous-domaines sont répartis équitablement entre tous les processus (256 sous-domaines associés à chaque processus, soit 64 Mo par processus). Dans ce cas, le temps d'une itération (c'est-à-dire la mise à jour de tous les sous-domaines) est de l'ordre de 2,5 secondes.

La figure 9.7 montre la proportion, par rapport à la reprise globale, de tâches à réexécuter et de processus impliqués lors d'un redémarrage en fonction de la période de sauvegarde. Les valeurs indiquées sont les valeurs dans le pire cas, c'est-à-dire quand la défaillance se produit juste avant la prochaine sauvegarde. Avec une période de sauvegarde de 60 secondes (soit environ 24 itérations), moins de 30 % des processeurs sont impliqués dans le redémarrage et seulement 6 % des tâches doivent être réexécutées.

Pour réduire le temps du redémarrage, on peut répartir l'ensemble des tâches à réexécuter sur tous les processeurs car il contient suffisamment de parallélisme (les 256 sous-domaines peuvent être distribués sur les autres processus). Le temps de calcul estimé pour réexécuter les tâches strictement nécessaires au redémarrage est de 3,6 secondes avec la reprise partielle contre 60 secondes avec la reprise globale. À ce temps, il faut ajouter le temps nécessaire pour calculer l'ensemble des tâches strictement nécessaires et le temps nécessaire pour distribuer les tâches et les données. Ces temps seront évalués par les expériences de la section 9.4.

9.3.1.3 Influence du nombre de processeurs

Les deux simulations suivantes montrent l'influence du nombre de processeurs sur la reprise partielle. La figure 9.8a présente la proportion de tâches à réexécuter par

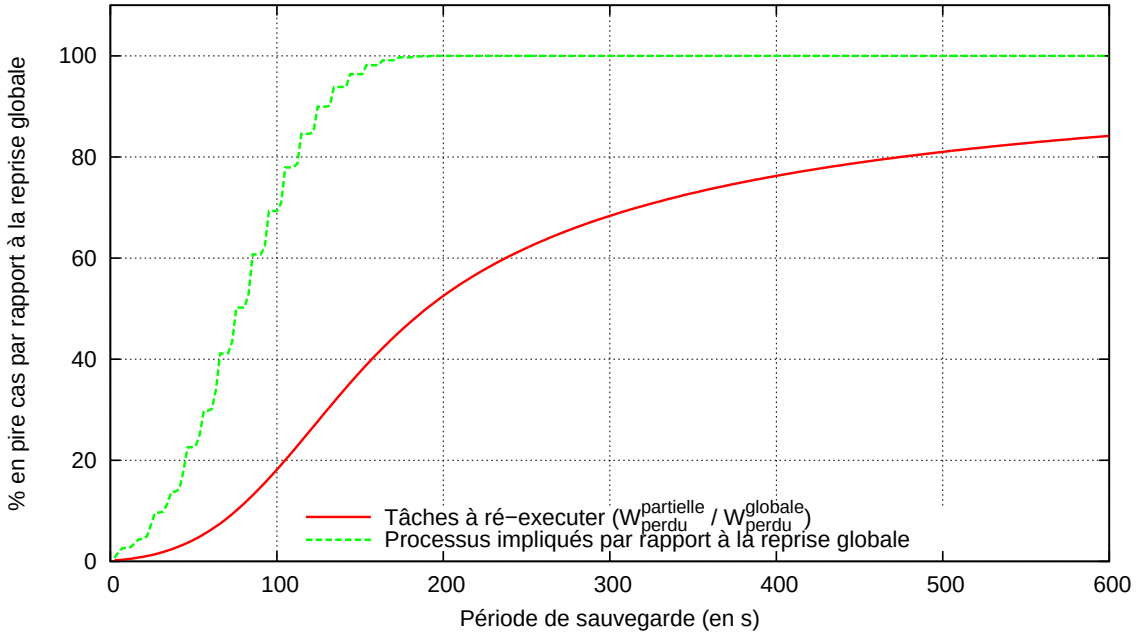


FIG. 9.7: Proportion en pire cas et par rapport à la reprise globale de tâches à réexécuter et de processus impliqués lors d'un redémarrage avec la reprise partielle

rapport à la reprise globale en fonction du nombre de processus et pour différentes périodes de sauvegarde.

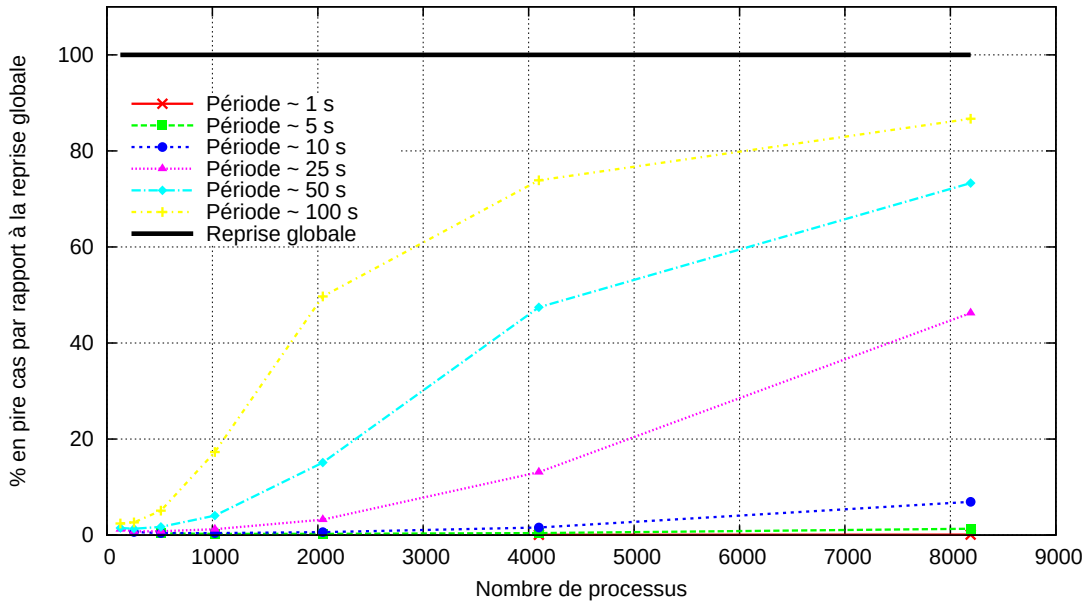
La figure 9.8b donne le nombre de processus impliqués lors d'un redémarrage par reprise partielle en fonction du nombre de processus et pour différentes périodes de sauvegarde.

Pour l'application décrite dans notre scénario exécutée sur 8 192 processeurs, une période de sauvegarde de 10 secondes permet d'obtenir moins de 10 % des tâches à réexécuter et moins de 2 500 processus impliqués (sur 8 192).

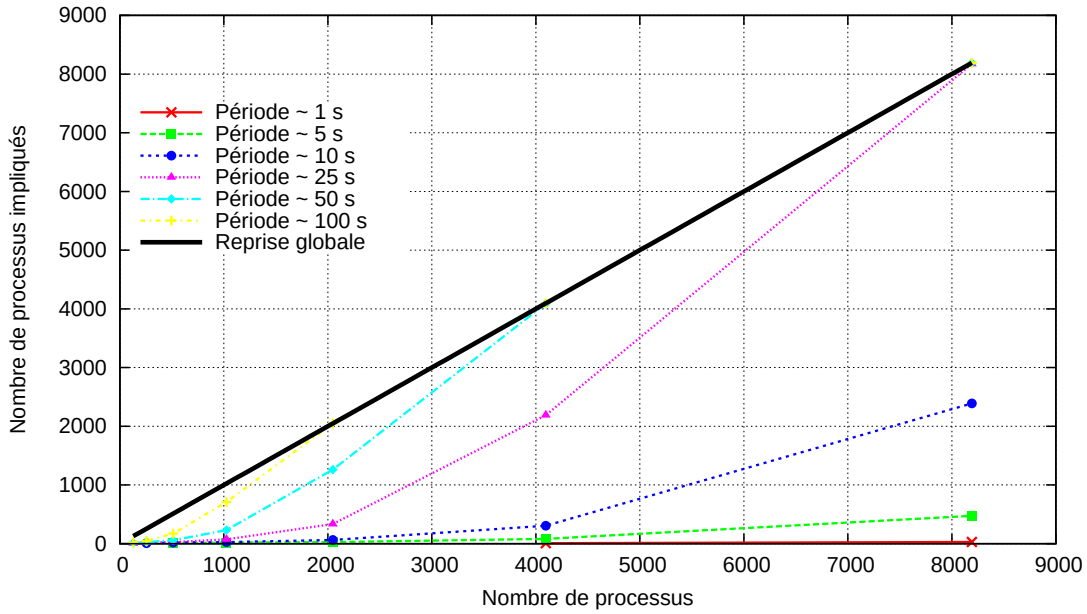
Entre deux sauvegardes, la quantité de calcul et le nombre d'itérations effectuées par l'application sont proportionnels au nombre de processeurs. Donc lorsque le nombre de processeurs augmente, la proportion du nombre de tâches à réexécuter et le nombre de processus impliqués augmentent car les graphes considérés sont plus grands et contiennent plus de dépendances. Pour conserver les bénéfices de la reprise partielle, il est nécessaire de diminuer la période de sauvegarde lorsque le nombre de processeurs augmente. De plus, cela permet de garantir qu'en cas de défaillance, le travail perdu ne sera pas trop important [67].

Ces simulations montrent l'intérêt de la reprise partielle en cas de défaillance. Néanmoins, un aspect, en pratique fondamental, n'a pas été pris en compte : les communications dues à la redistribution des données.

La principale raison est qu'il aurait été difficile d'avoir des résultats de simulations qui puissent être liés aux résultats des mesures expérimentales que nous allons présenter dans la section suivante. En effet, la diversité de matériel, de configuration, de réseaux, etc. de la plateforme Grid'5000 font que cette difficulté aurait été à la fois conceptuelle,



(a) Proportion de tâches à réexécuter par rapport à la reprise globale



(b) Nombre de processus impliqués

FIG. 9.8: Proportion de tâches à réexécuter et nombre de processus impliqués pour la reprise partielle en fonction du nombre de processeurs et pour différentes périodes de sauvegarde

due à la modélisation d'une telle plateforme, et pratique, car consommatrice de temps au détriment d'expérimentations sur une vraie architecture.

9.4 Expérimentations

Nous avons réalisé trois séries d'expériences pour évaluer notre protocole de reprise partielle. Tout d'abord, la première série d'expériences vise à comparer les simulations de la section 9.3.1.2 à des redémarrages sur une véritable application. Ensuite, nous étudions le cout de notre algorithme qui détermine l'ensemble des tâches nécessaires pour reprendre l'exécution. Enfin, nous étudions le temps nécessaire pour réexécuter le travail perdu pour la reprise partielle et nous le comparons à la reprise globale.

Ces expériences ont été réalisées sont plusieurs sites de Grid'5000 en utilisant l'application de décomposition de domaine qui a également servi aux sections expérimentales des chapitres 7 et 8.

9.4.1 Influence de la période de sauvegarde

À travers cette expérience, nous cherchons à évaluer la proportion de tâches à réexécuter pour la reprise partielle par rapport à la reprise globale. Cette valeur dépend de beaucoup de paramètres :

- de l'application, en particulier du schéma de dépendances qui indique les sous-domaines voisins utilisés pour mettre à jour un sous-domaine ;
- de la forme du domaine de calcul, c'est-à-dire son rapport entre les longueurs de ses dimensions (largeur et longueur dans le cas d'un domaine en deux dimensions) ;
- du nombre de machines utilisées à l'exécution et du niveau de sur-décomposition employé ;
- du nombre de machines défaillantes et des sous-domaines qui leur sont associés ;
- de la période de sauvegarde, ou plutôt de la quantité de travail exécuté entre la dernière sauvegarde et la panne.

Pour étudier ce phénomène, nous allons nous limiter en fixant tous les paramètres excepté la période de sauvegarde que nous allons faire varier. Nous considérons l'application de décomposition de domaine dont le modèle est présenté dans la section 8.4.1 avec un domaine total découpé en $40 \times 40 \times 1 = 1600$ sous-domaines. Nous réalisons une exécution sur 100 machines du site de Nancy de Grid'5000 ; chaque machine possède donc 16 sous-domaines. Nous simulons alors la panne d'une machine fixée et nous mesurons, pour différentes valeurs de la période de sauvegarde, la proportion de tâches à réexécuter, en pire cas, par rapport à l'ensemble total des tâches qui devraient être réexécutées dans le cas de la reprise globale. Le pire cas correspond au cas où la panne se produit juste avant la prochaine sauvegarde.

La figure 9.9 montre le résultat de ces mesures. Nous avons également tracé sur cette figure le résultat d'une simulation pour des conditions identiques aux mesures. Les deux courbes ont des allures similaires. On constate une différence maximale proche de 15 % autour de l'itération 50.

Les différences entre la simulation et les mesures expérimentales s'expliquent par le fait que le modèle utilisé pour les simulations ne correspond pas exactement au cas réel. Les principales différences sont que :

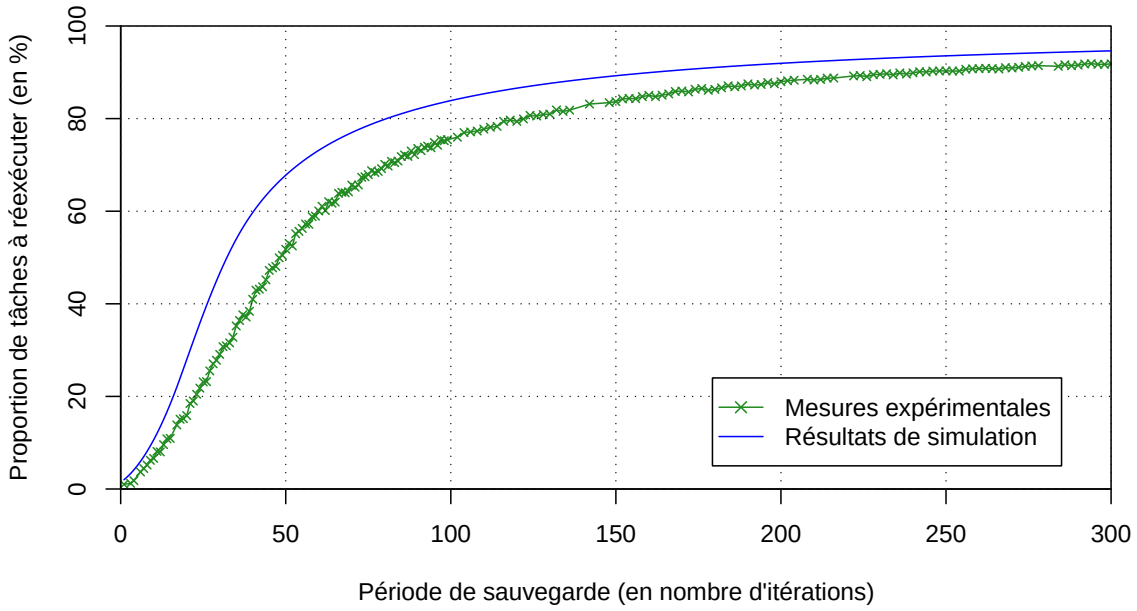


FIG. 9.9: Proportion de tâches à réexécuter en pire cas pour la reprise partielle par rapport à la reprise globale, en fonction de la période de sauvegarde

- la simulation utilise un modèle simplifié de l'application (similaire à celui présenté en 8.4.1) qui comporte moins de tâches que la véritable application ;
- le modèle de la simulation, contrairement à l'application réelle, utilise un domaine torique où les sous-domaines extrêmes sont voisins, ceci dans le but de simplifier le choix du processus défaillant.

Nous concluons cette expérience en remarquant que les simulations nous ont permis d'obtenir une impression assez juste de l'évolution de la proportion de tâches à réexécuter dans le cas de la reprise partielle.

9.4.2 Cout de l'algorithme de reprise partielle

Par cette expérience, nous étudions le cout en temps de calcul de l'algorithme qui détermine l'ensemble des tâches nécessaires à la reprise partielle.

L'algorithme présenté à la section 9.2.3 de ce chapitre est un algorithme distribué. Pour des raisons de simplicité, la version de cet algorithme implémentée dans KAAPI est centralisée.

La figure 9.10 a été réalisée en mesurant le temps d'exécution de l'implémentation centralisée de l'algorithme. Comme l'étude de la section 9.2.3.2 l'a montré, le cout en temps de cet algorithme est proportionnel au nombre de tâches contenues dans le graphe de flot de données de l'application.

Pour un nombre de tâches inférieur à 1 million, le temps d'exécution de l'algorithme est inférieur à 2,5 secondes ; ce qui est un temps raisonnable comparé au temps qui sera nécessaire pour réexécuter le travail perdu (*cf* section suivante).

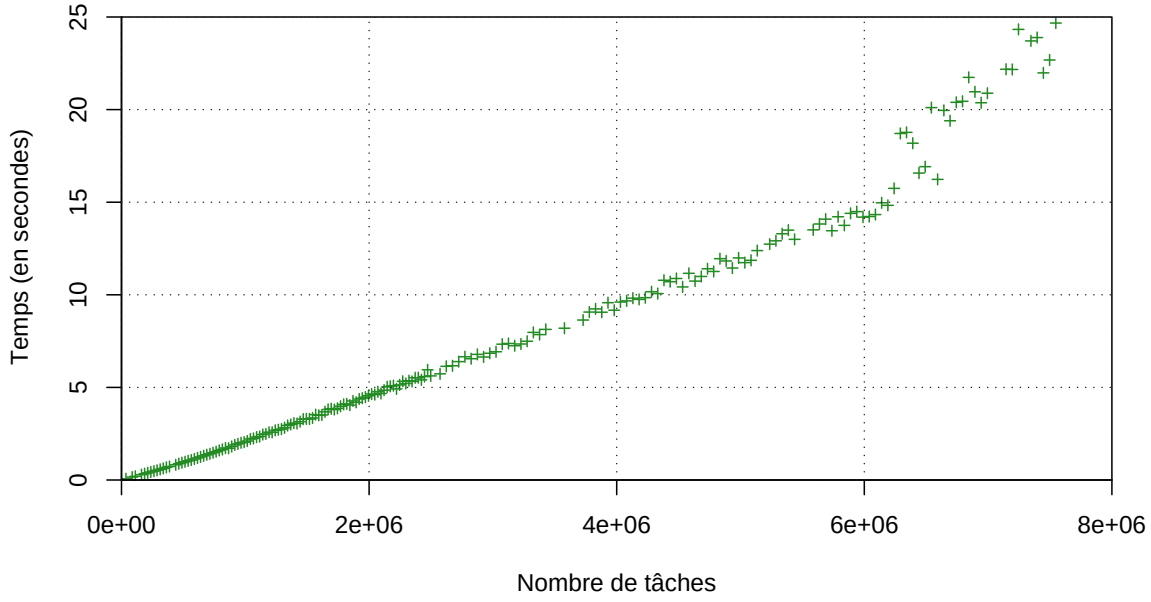


FIG. 9.10: Cout de l'algorithme qui calcule l'ensemble des tâches nécessaires à la reprise partielle en fonction du nombre de tâches du graphe de flot de données

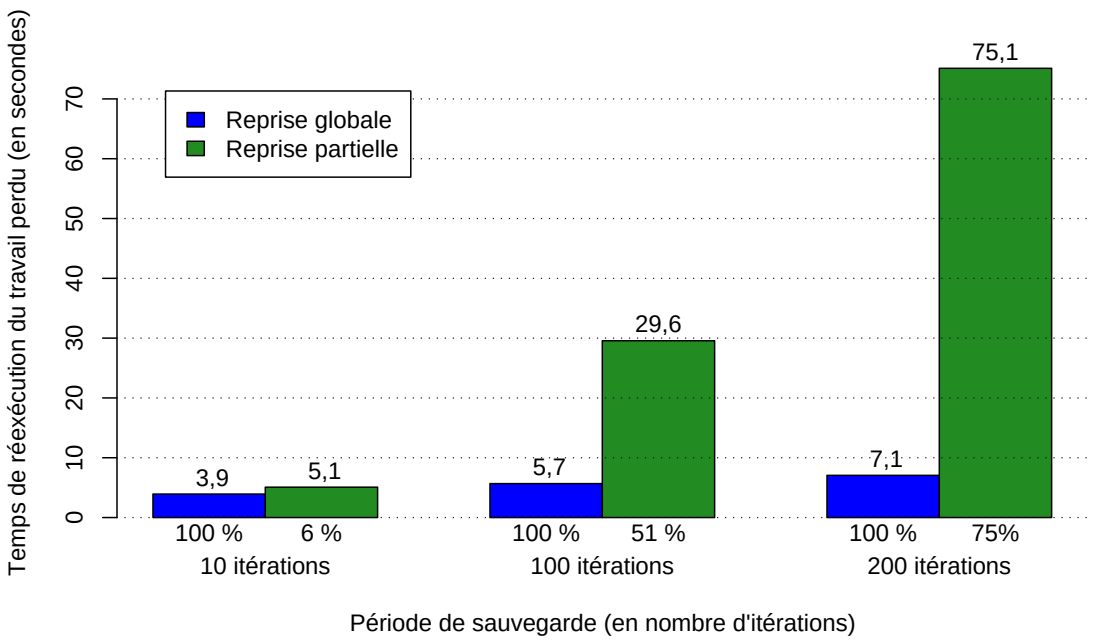
À partir de 6 millions de tâches, les temps mesurés deviennent très irréguliers. Ceci est dû au fait que pour cette taille de graphe, les 2 Go de mémoire de la machine sur laquelle ont été faites les mesures étaient saturées. L'implémentation d'une version distribuée de l'algorithme (*cf* section 9.2.3) permettrait de régler ce problème.

9.4.3 Temps de réexécution du travail perdu

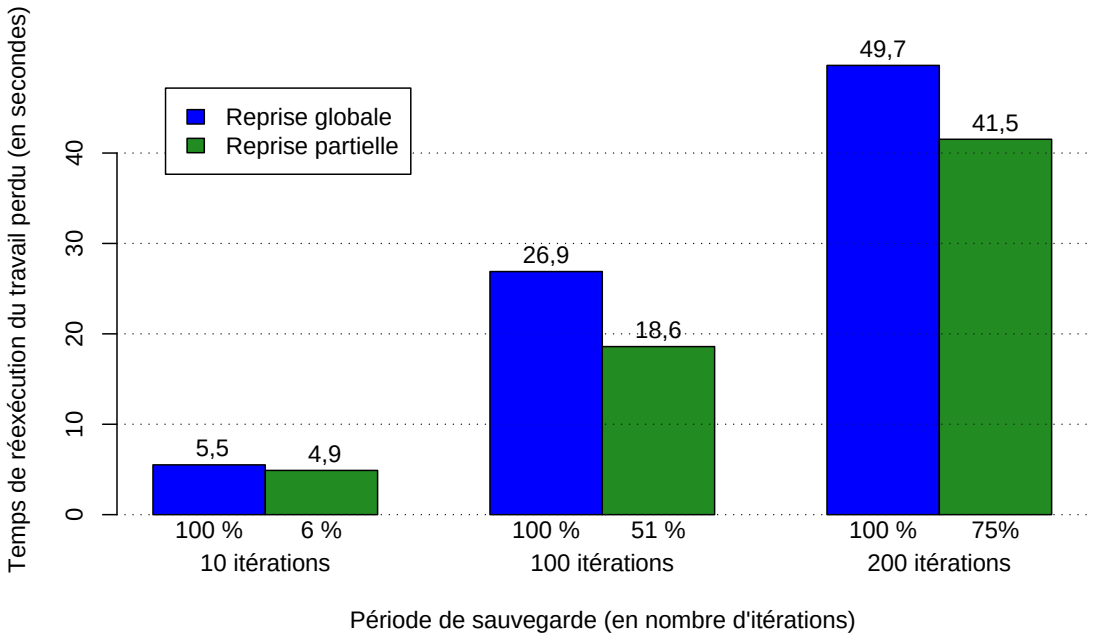
Pour cette dernière série d'expériences, nous étudions le temps de réexécution du travail perdu pour notre protocole de reprise partielle et nous le comparons à celui de la reprise globale. Comme nous l'avons expliqué pour les mesures de la section 8.5.3 et pour des raisons d'implémentation, le temps $T_{réexécution}$ mesuré prend en compte à la fois le temps de réexécution du travail perdu et également la redistribution des données nécessaires à cette réexécution.

Les mesures de la figure 9.11 ont été réalisées sur 110 machines de Bordeaux de Grid'5000. 100 machines ont été utilisées pour les processus de calcul et 10 machines pour les serveurs de sauvegarde. Nous utilisons la même application de décomposition de domaine avec un domaine total de 76 Mo.

Nous faisons varier la période de sauvegarde et nous simulons la panne d'une machine en pire cas, c'est-à-dire juste avant la prochaine sauvegarde. La période de sauvegarde indique donc le nombre d'itérations qui ont été calculées entre la dernière sauvegarde et la panne. Nous avons réalisé des mesures pour des périodes de sauvegarde de 10, 100 et 200 itérations. Pour le domaine de calcul utilisé, les proportions de tâches à réexécuter

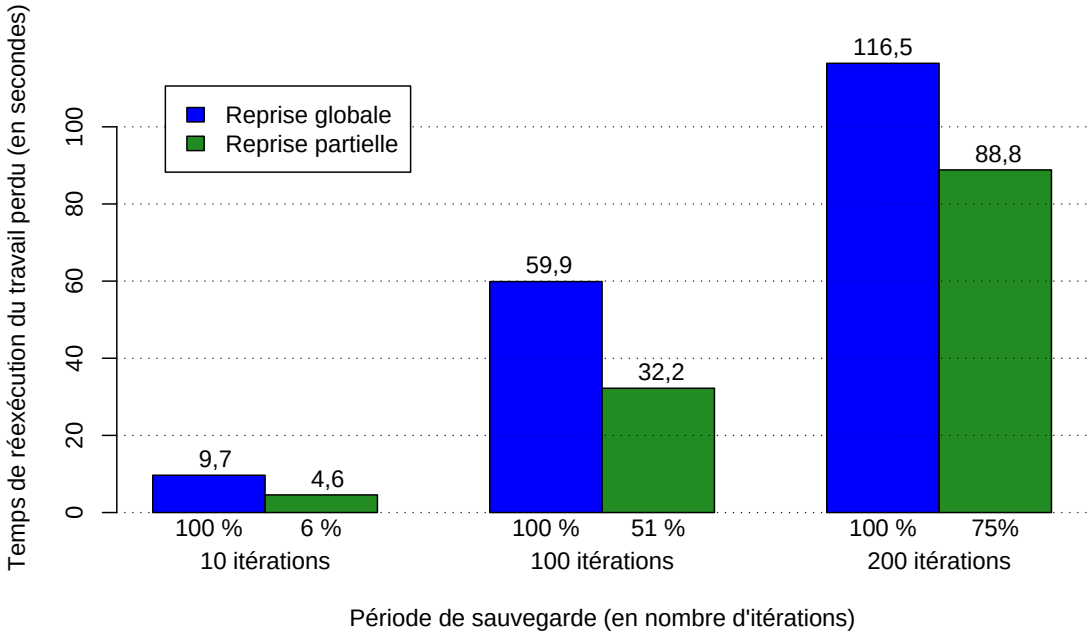


(a) Rapport calcul/communication × 1

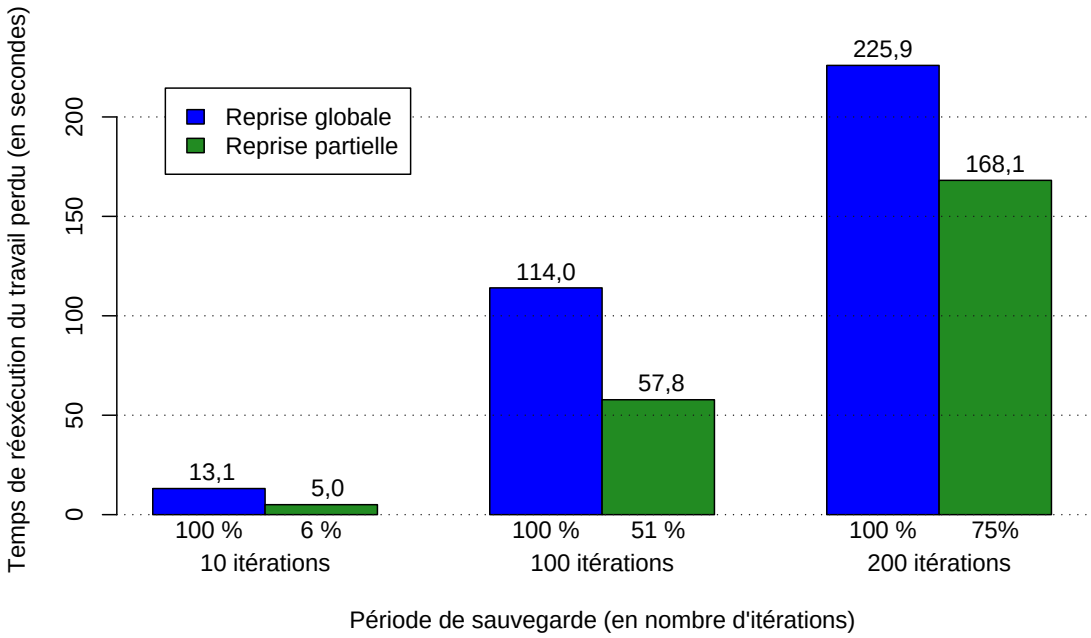


(b) Rapport calcul/communication × 10

FIG. 9.11: Comparaison du temps de réexécution du travail perdu entre la reprise globale et la reprise partielle pour différents grains



(c) Rapport calcul/communication × 25



(d) Rapport calcul/communication × 50

FIG. 9.11: Comparaison du temps de réexécution du travail perdu entre la reprise globale et la reprise partielle pour différents grains

pour la reprise partielle par rapport à la reprise globale sont respectivement de 6 %, 51 % et 75 %.

Dans le but de simuler des applications qui ont un ratio calcul/communication plus important, nous adaptons le code de calcul de notre application de décomposition de domaine de manière à changer le grain de calcul de la tâche de mise à jour d'un sous-domaine. Pour cette expérience, ce grain de référence vaut 0,002 seconde. Le cas (a) correspond au grain de référence ; la mise à jour d'un point du domaine est réalisée en calculant une simple combinaison linéaire des points voisins. Les cas (b), (c) et (d) correspondent respectivement à des grains de 10, 25 et 50 fois le grain de référence. Ils permettent de simuler des applications avec des opérations de mise à jour du domaine plus complexes.

Dans le cas de la sous figure 9.11a, nous constatons que, bien que la quantité de travail à réexécuter soit plus faible (seulement 6 %, 51 % ou 75 %), le temps de réexécution du travail perdu est bien plus important dans le cas de la reprise partielle. Cette différence importante est due à la redistribution des données qui est indispensable pour rééquilibrer le travail à la reprise.

En effet, dans le cas de la reprise partielle, le nombre de tâches à réexécuter est plus faible. Lors de la phase d'équilibrage de charge de la reprise, ces tâches sont plus éparpillées que pour une reprise globale. En conséquence, une grande partie des données doit être redistribuée et cela induit un surcout important.

C'est l'étape de partitionnement statique (*cf* section 4.3.2.3) qui est chargée d'équilibrer la charge de calcul à la reprise. Le travail est réparti dans un ensemble de K-threads et un algorithme d'équilibrage de charge est utilisé pour déterminer le placement de K-threads sur toutes les machines disponibles.

Pour la reprise globale, l'algorithme d'équilibrage de charge utilisé est basé sur une méthode de répartition en bloc-cyclique. Cette méthode est assez performante puisque, pour le reprise globale, les K-threads à répartir représentent des quantités de calcul équivalentes et sont en nombre suffisant grâce à la sur-décomposition.

Pour la reprise partielle, ces considérations ne sont plus vraies. Le partitionnement statique utilise un algorithme d'équilibrage de charge basé sur LPT (*Longest Processing Time*). La quantité de travail à réexécuter étant plus faible et déséquilibrée entre les K-threads, cet algorithme permet d'obtenir un chemin critique moins important. Cependant, il ne prend pas du tout en compte le placement des données et les communications entre les K-threads générés.

Les mauvaises performances de la reprise partielle sur le cas de la figure 9.11a sont donc causées par une mauvaise répartition du travail à la reprise qui ne prend pas en compte le placement des données d'avant la panne. Cette expérience met en évidence le fait que la répartition de la charge et le placement des tâches de calcul sont des éléments critiques pour obtenir de bonnes performances à la reprise et, ceci de manière beaucoup plus importante avec la reprise partielle.

Sur les figures 9.11b, 9.11c et 9.11d, nous avons respectivement augmenté le ratio calcul/communication d'un facteur 10, 25 et 50. L'augmentation du ratio calcul/communication permet de masquer le cout de redistribution des données par rapport au temps de calcul des tâches à réexécuter.

Pour une augmentation d'un facteur 10, sur la figure 9.11b, la reprise partielle permet d'obtenir un temps de réexécution du travail perdu plus faible que pour la reprise globale.

Pour une augmentation d'un facteur 25 ou 50, sur les figures 9.11c et 9.11d, et pour les périodes de sauvegarde de 100 et 200 itérations, le rapport des temps de réexécution $T^{partielle}_{réexécution}/T^{globale}_{réexécution}$ est proche de la proportion de tâches à réexécuter calculée, soit 51 % pour une période de 100 itérations et 75 % pour une période de 200 itérations.

Pour une faible période de sauvegarde, 10 itérations sur la figure, la reprise partielle est seulement deux fois plus rapide. Cela s'explique par le fait que le cout de redistribution des données reste trop important pour obtenir le rapport théorique de 6 %.

9.5 Conclusion

Dans ce chapitre, nous avons proposé un protocole original de reprise partielle, appelé CCK-Restart. Ce protocole utilise le graphe de flot de données de l'application et les dépendances entre les tâches pour déterminer l'ensemble des tâches strictement nécessaires à la reprise du calcul.

Nous avons présenté ce protocole et nous l'avons étudié théoriquement. Nous avons également réalisé plusieurs simulations qui permettent d'évaluer le gain obtenu par rapport au protocole de reprise globale présenté dans le chapitre précédent.

Enfin, nous avons expérimenté ce protocole de reprise partielle sur une application réelle de décomposition de domaine. Deux conclusions majeures ressortent de ces expérimentations.

- Tout d'abord, le gain de ce protocole de reprise partielle en termes de quantité de tâches à réexécuter peut être important. De plus, pour l'application considérée, les simulations ont permis d'estimer la proportion de tâches à réexécuter de manière satisfaisante.
- Ensuite, la redistribution des données induite par le rééquilibrage de charge à la reprise a une influence considérable sur le temps de réexécution du travail perdu, en particulier lorsque le rapport calcul/communication est faible. L'utilisation d'algorithme d'équilibrage de charge prenant en compte à la fois le cout des tâches et le placement initial des données permettra d'améliorer les performances de ce protocole.

Conclusion et perspectives 10

Sommaire

10.1 Bilan

Dans cette thèse, nous avons abordé les aspects de la reconfiguration dynamique et de la tolérance aux fautes pour les applications distribuées sur les architectures de type grille de calcul.

Tout d'abord, nous avons proposé un mécanisme de reconfiguration dynamique pour l'environnement de programmation parallèle Kaapi. Comme la majorité des travaux présentés dans l'état de l'art (chapitre 3), ce mécanisme est basé sur une représentation abstraite de l'état de l'application. Dans notre cas, cette représentation est le graphe de flot de données fourni par le moteur exécutif Kaapi. Elle nous permet notamment d'inspecter l'état et de modifier le comportement de l'application en cours d'exécution.

Le mécanisme de reconfiguration que nous avons conçu porte sur deux aspects de la reconfiguration : la gestion des accès concurrents et la gestion de la cohérence.

La gestion des accès concurrents permet de protéger la reconfiguration des modifications externes pouvant intervenir durant son exécution. Bien que cet aspect soit abordé dans la littérature sur des reconfigurations spécifiques (par exemple le vol de travail [14, 50, 141]), nous ne connaissons pas de travaux qui l'aient étudié dans le cadre général de la reconfiguration dynamique.

Nous avons défini deux méthodes d'exécution des reconfigurations : l'exécution concurrente et l'exécution coopérative. L'exécution coopérative permet de réduire l'utilisation de primitives de synchronisation dans l'implémentation du mécanisme de reconfiguration. Ceci est possible au prix d'une latence plus importante pour l'exécution de la reconfiguration, principalement lors de calcul avec un gros grain. Cette technique a été intégrée au logiciel X-Kaapi et elle a montré de très bons résultats à grain fin en comparaison de TBB et Cilk.

La gestion de la cohérence vise à garantir un état correct de l'application après une reconfiguration. En particulier, la cohérence mutuelle est l'aspect qui permet d'offrir au programme de reconfiguration une vision cohérente d'un ensemble d'objets distribués.

Du point de vue général de la reconfiguration dynamique, cet aspect a été étudié à de nombreuses reprises et notamment avec les logiciels présentés dans le chapitre 3. Dans la plupart des cas, une représentation abstraite de l'application (généralement sous forme d'un graphe) est utilisée pour assurer cette propriété de manière transparente pour l'utilisateur.

Nous avons tout d'abord défini les propriétés de cohérence et d'accessibilité de l'état global qui permettent d'assurer la cohérence mutuelle entre des objets distribués. Ensuite, nous avons proposé pour Kaapi un protocole optimisé de gestion de la cohérence mutuelle qui repose sur la connaissance du graphe de flot de données de l'application pour réduire le cout de la coordination. Nous avons montré expérimentalement les bonnes performances de ce protocole (rapidité et faible variabilité du temps d'exécution) sur un millier de machines de la plateforme Grid'5000.

Dans le cadre de la tolérance aux fautes, nous avons principalement étudié le protocole classique de sauvegarde/reprise coordonnée. Nous avons notamment intégré ce protocole dans le logiciel Kaapi en réutilisant les mécanismes de reconfiguration proposés précédemment. Les applications ciblées sont les applications itératives de type décomposition de domaine. Nous avons proposé plusieurs améliorations de ce protocole en nous basant sur le graphe de flot de données représentant l'application.

Une première amélioration permet d'éviter le ralentissement de la vitesse d'exécution de l'application après la reprise lorsqu'il n'y a pas de machine de rechange disponible. Elle repose à la fois sur la sur-décomposition du domaine de calcul de l'application et sur le rééquilibrage du calcul à la reprise. Les résultats expérimentaux confirment la pertinence de cette approche tant que le niveau de sur-décomposition n'est pas trop important, auquel cas le surcout de gestion des tâches pénalise l'éxécution.

Une seconde amélioration est la conception d'un protocole original de reprise partielle basé sur la sauvegarde coordonnée. Grâce au graphe de flot de données de l'application, ce protocole détermine l'ensemble des tâches strictement nécessaires pour redémarrer l'application dans un état correct. Selon les conditions, il peut réduire considérablement la quantité de travail perdu à réexécuter par rapport à la reprise globale. Les expérimentations confirment ces gains mais mettent en évidence le surcout important lié à la redistribution des données. Ce cout est principalement dû au fait que l'algorithme de rééquilibrage de charge ne tient pas compte du placement des données avant la défaillance.

Le bilan de ces travaux de recherche s'exprime aussi en termes de développement logiciel. Dans l'environnement Kaapi qui représente à lui seul plus de 100 000 lignes de C++, ces contributions portent principalement sur deux modules.

- Le module de tolérance aux fautes de Kaapi contient l'implémentation de tous les mécanismes et les composants nécessaires (*cf* section 7.2) pour exécuter une application de manière transparente en présence de fautes. Les développements dans ce module représentent 10 500 lignes de code.
- Le module d'ordonnancement par partitionnement statique contient les mécanismes qui permettent l'exécution efficace d'applications itératives à grande échelle. Il a été développé conjointement avec Laurent Pigeon dans le cadre sa thèse [117] et constitue environ 10 000 lignes de codes.

Enfin, l'utilisation et l'expérimentation à grande échelle sur Grid'5000 ont permis de mettre en évidence certains défauts de KAAPI et de les corriger. Cela a abouti, entre autres, à l'outil de déploiement `karun` qui repose sur le logiciel TAKTUK [54] et, à la mise en place de bonnes pratiques pour l'expérimentation. Le savoir-faire est conséquent (deux victoires aux PlugTests organisés par l'ETSI en 2007 et 2008), mais il faut remarquer que le développement logiciel et l'expérimentation à grande échelle nécessitent un temps considérable.

10.2 Perspectives

Nous présentons maintenant plusieurs perspectives de recherche qui peuvent être explorées suite à nos travaux. Après des perspectives à court terme, nous présentons trois axes de recherche liés à notre connaissance des problèmes de tolérance aux fautes et aux possibilités de reconfigurer les applications dynamiquement.

10.2.1 Perspectives à court terme

De l'ensemble de travaux effectués, nous avons dégagé deux perspectives de recherche à court terme : la première vise à étendre le cadre d'application de X-KAAPI pour des applications parallèles ; la seconde concerne la poursuite des comparaisons expérimentales des performances des protocoles de tolérance aux fautes.

X-Kaapi. Les performances obtenues avec X-KAAPI dans le cadre d'une exécution coopérative de l'opération de reconfiguration « vol de travail » sont nettement supérieures à celles observées avec TBB ou CILK. Néanmoins, elles nécessitent que l'application intervienne dans le traitement des requêtes de vol. Il est donc important de chercher à combiner les deux approches afin que le moteur exécutif puisse profiter d'une exécution coopérative lorsque l'application le peut tout en autorisant une exécution concurrente si l'application est occupée par son calcul.

Des travaux dans cette direction ont commencé dans le projet MOAIS. À terme, le moteur X-KAAPI devrait remplacer le moteur exécutif de KAAPI.

Comparaisons expérimentales. À court terme et en complément direct à nos travaux, il serait intéressant d'étendre les expérimentations et les comparaisons que nous avons réalisées à d'autres architectures et à d'autres logiciels.

Les solutions présentées dans cette thèse visaient une architecture de type grille de calcul. Nous avons pu mesurer leurs performances en utilisant jusqu'à un millier de machines. Cependant, leur comportement sur des grilles de plus grande taille ou sur des supercalculateurs n'est pas connu. De même, il est important de réaliser une comparaison des performances des protocoles de tolérance aux fautes de KAAPI avec ceux d'autres environnements comme OPENMPI ou CHARM++/AMPI. L'objectif étant d'identifier les éléments clés dans l'obtention de performances. De ce point de vue, nous pensons que l'utilisation d'une représentation abstraite sous la forme d'un graphe de flot de donnée est un élément fondamental permettant beaucoup d'optimisations. Il reste à le démontrer par l'expérience. Une partie de ces travaux sera réalisée au mois de

mars 2010 durant une visite dans le *Parallel Programming Laboratory* de l'Université d'Urbana-Champaign.

10.2.2 Tolérance aux fautes : difficultés et perspectives

Bien que le domaine de la tolérance aux fautes soit étudié depuis longtemps, beaucoup de problèmes restent ouverts.

Mémoire stable. Du point de vue de la sauvegarde, le problème essentiel est lié au surcout important induit par les sauvegardes. Hormis le fait de réduire le cout de la sauvegarde, par exemple en réduisant le volume des données sauvegardées, le choix de dates de sauvegarde adéquates permettrait de diminuer l'impact de la sauvegarde sur l'exécution de l'application. De nombreux travaux existent déjà dans ce domaine [163, 41], mais les solutions proposées se basent sur des modèles qui dépendent de paramètres du système généralement inconnus en pratique (loi de distribution d'apparition des pannes, taille de la sauvegarde, durée de la reprise, etc.) et elles ne sont donc pas ou peu utilisées. Ainsi, la conception d'algorithmes « en ligne » de choix des intervalles de sauvegarde nous semble intéressant. Les paramètres manquant seraient alors approchés dynamiquement à l'exécution.

Une autre direction est l'utilisation d'un service distribué pour la réalisation d'une mémoire stable en utilisant des codes correcteurs pour permettre la reconstruction des données sauvegardées même en cas de défaillance des serveurs. Bien que cette possibilité ait été étudiée il y a maintenant 14 ans [119] et dont certains éléments sont disponibles à travers un logiciel [121], aucun des logiciels de calcul haute performance ne se base sur ce genre de technique. Le manque d'expérimentations dans ce cadre ne permet pas de conclure sur l'intérêt de l'approche et il serait intéressant de reprendre ce travail sur les architectures actuelles.

Protocole de reprise. Du point de vue de la reprise, nous avons vu que le premier facteur dégradant les performances, plus particulièrement dans le cas de la reprise partielle, est la redistribution des données induite par le rééquilibrage de charge. Pour limiter ce surcout et diminuer le temps de reprise, il est donc nécessaire de réaliser l'équilibrage de charge en prenant en compte à la fois le cout de calcul des tâches, mais aussi le placement des données, qu'elles soient stockées sur les processus de calcul ou sur les serveurs de sauvegarde.

10.2.3 Reconfiguration dynamique et exécution autonome

Les mécanismes de reconfiguration présentés dans le chapitre 5 se veulent génériques. Cependant, au cours de ces travaux, ils ont seulement été appliqués à l'ordonnancement (vol de travail et partitionnement statique) et aux protocoles de tolérance aux fautes. Nous proposons ici deux autres cadres pour lesquels ces mécanismes pourraient être utilisés.

Couplage de code et débogage d'applications distribuées. De nombreuses reconfigurations sont imaginables comme le couplage de code dans le but de récupérer, à un instant donné, un état cohérent de l'application afin de le visualiser [70]. De cette manière, il serait alors possible de coupler, de manière transparente, KAAPI à l'outil de visualisation FLOWVR [103]. L'interêt d'utiliser la mécanique de reconfiguration de KAAPI est de ne pas mélanger le code de visualisation et le code de calcul.

Ce mécanisme de reconfiguration pourrait aussi être utilisé pour réaliser du débogage d'applications distribuées. Par exemple, la consultation d'une donnée consisterait à ajouter dynamiquement une tâche qui affiche sa valeur. Combiné à un protocole de sauvegarde, il serait possible de faire revenir l'exécution en arrière pour reproduire facilement un bogue.

Exécution autonome. Enfin, le mécanisme de reconfiguration dynamique proposé pour KAAPI peut être utilisé pour réaliser de l'adaptation dynamique[1]. Pour cela, il manque encore la réalisation des fonctions d'observation et de décision vues dans le chapitre 3. Les adaptations visées seront alors, l'ajout et le retrait dynamique de machines afin d'offrir des garanties de performances d'une manière similaire à SATIN [158] : si l'efficacité du calcul est suffisante, des machines peuvent être ajoutées, si l'efficacité est mauvaise des machines doivent être retirées.

Dans un but similaire, on peut envisager une reconfiguration qui transforme le calcul, par exemple en changeant la méthode numérique par la transformation du graphe de tâches représentant l'exécution.

[1] C'est même le but initial.

Bibliographie

[1] Vikram Adve, Vinh Vi Lam et Brian Ensink : Language and compiler support for adaptive distributed applications. *In LCTES '01 : Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 238–246, New York, NY, USA, 2001. ACM.

[2] Adnan Agbaria et Roy Friedman : Starfish : Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, juillet 2003.

[3] Marco Aldinucci, Sonia Campa, Massimo Coppola, Marco Danelutto, Domenico Laforenza, Diego Puppin, Luca Scarponi, Marco Vanneschi et Corrado Zoccolo : Components for high performance grid programming in Grid.it. *In Workshop on Component Models and Systems for Grid Applications, CoreGRID series*. Springer, 2005.

[4] Marco Aldinucci, Massimo Coppola, Marco Danelutto, Marco Vanneschi et Corrado Zoccolo : ASSIST as a research framework for high-performance grid programming environments. *In Grid Computing : Software environments and Tools*, pages 230–256. Springer, 2005.

[5] Marco Aldinucci, Ro Petrocelli, Edoardo Pistoletti, Massimo Torquati, Marco Vanneschi, Luca Veraldi et Corrado Zoccolo : Dynamic reconfiguration of grid-aware applications in ASSIST. *In Euro-Par 2005 Parallel Processing*, volume 3648 de *Lecture Notes in Computer Science*, pages 771–781, Lisbon, Portugal, août 2005. Springer.

[6] João P. A. Almeida : Dynamic reconfiguration of object-middleware-based distributed systems. Master's thesis, University of Twente, The Netherlands, 2001.

[7] João P. A. Almeida, Maarten Wegdam, Luis F. Pires et Marten van Sinderen : An approach to dynamic reconfiguration of distributed systems based on object-middleware. *CTIT technical reports series*, 1(6), 2001.

[8] Lorenzo Alvisi et Keith Marzullo : Message logging : Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[9] Lorenzo Alvisi, Sriram Rao, Syed Amir Husain, Asanka De Mel et Elmootazbellah N. Elnozahy : An analysis of communication-induced checkpointing. *International Symposium on Fault-Tolerant Computing*, 0:242, 1999.

[10] Noriki Amano et Takuo Watanabe : An approach for constructing component-based software systems with dynamic adaptability using LEAD++. *In International Symposium on Principles of Software Evolution, 2000*, pages 118–127, 2000.

[11] David P. Anderson : BOINC : A system for public-resource computing and storage. *IEEE/ACM International Workshop on Grid Computing*, 0:4–10, 2004.

[12] Jason Ansel, Kapil Arya et Gene Cooperman : DMTCP : Transparent checkpointing for cluster computations and the desktop. *In International Parallel and Distributed Processing Symposium*, Rome, Italy, mai 2009.

[13] Jean Arlat, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Jean-Claude Laprie et David Powell : *Encyclopédie de l'Informatique et des Systèmes d'Information*, chapitre Tolérance aux fautes, pages 241–270. Vuibert, 2006.

[14] Nimar S. Arora, Robert D. Blumofe et C. Greg Plaxton : Thread scheduling for multiprogrammed multiprocessors. *In SPAA '98 : Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, New York, NY, USA, 1998. ACM.

[15] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mark A. Taylor, Timothy S. Woodall et Mitchel W. Sukalski : Architecture of LA-MPI, a network-fault-tolerant MPI. *Parallel and Distributed Processing Symposium, International*, 1:15b, 2004.

[16] Algirdas Avizienis : Fault-tolerant systems. *IEEE Transactions on Computers*, 25(12):1304–1312, 1976.

[17] Algirdas Avizienis, Jean-Claude Laprie et Brian Randell : Dependability and its threats - a taxonomy. *In IFIP Congress Topical Sessions*, pages 91–120, 2004.

[18] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell et Carl E. Landwehr : Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[19] Roberto Baldoni, Jean-Michel Helary, Achour Mostefaoui et Michel Raynal : A communication-induced checkpointing protocol that ensures rollback-dependency trackability. 0:68, 1997.

[20] Rajanikanth Batchu, Anthony Skjellum, Zhenqian Cui, Murali Beddhu, Jothi P. Neelamegam, Yoginder Dandass et Manoj Apte : MPI/FT : Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. *IEEE International Symposium on Cluster Computing and the Grid*, 0:26, 2001.

[21] Francoise Baude, Denis Caromel, Christian Delbé et Ludovic Henrio : A hybrid message logging-cic protocol for constrained checkpointability. *In Euro-Par 2005 Parallel Processing*, volume 3648 de *Lecture Notes in Computer Science*, pages 644–653, Lisbon, Portugal, août 2005. Springer.

[22] Olivier Beaumont, El Mostafa Daoudi, Nicolas Maillard, Pierre Manneback et Jean-Louis Roch : Tradeoff to minimize extra-computations and stopping criterion tests for parallel iterative schemes. *In 3rd International Workshop on Parallel Matrix Algorithms and Applications (PMAA04)*, CIRM, Marseille, France, octobre 2004.

[23] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-crummey, Dan Reed, Linda Torczon et Rich Wolski : The

GrADS project : Software support for high-level grid application development. *International Journal of High Performance Computing Applications*, 15:327–344, 2001.

[24] Francine Berman, Rich Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil T. Spring, Alan Su et Dmitrii Zagorodnov : Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[25] Marin Bertier, Olivier Marin et Pierre Sens : Performance analysis of a hierarchical failure detector. *In Proceedings of the International Conference on Dependable Systems and Networks*, pages 635–644, 2003.

[26] Xavier Besseron : IV Grid Plugtests : composing dedicated tools to run an application efficiently on Grid'5000. Communication orale, février 2008. 3rd EGEE User Forum.

[27] Xavier Besseron : Optimized coordinated checkpoint/rollback protocol using a dataflow graph model. Communication orale, janvier 2009. Workshop APRETAF : Algorithmes Parallèles, Répartis Et Tolérance Aux Fautes.

[28] Xavier Besseron, Mohamed Slim Bouguerra, Thierry Gautier, Érik Saule et Denis Trystram : *Fault tolerance and availability awareness in computational grids*, chapitre 5. Numerical Analysis and Scientific Computing. Chapman and Hall/CRC Press, 2009.

[29] Xavier Besseron et Thierry Gautier : Optimised recovery with a coordinated checkpoint/rollback protocol for domain decomposition applications. *In Modelling, Computation and Optimization in Information Systems and Management Sciences (MCO '08)*, pages 497–506, Metz, France, septembre 2008. Springer.

[30] Xavier Besseron, Samir Jafar, Thierry Gautier et Jean-Louis Roch : CCK : An improved coordinated checkpoint/rollback protocol for dataflow applications in Kaapi. *In ICTTA '06 IEEE Conference on Information and Communication Technologies : from Theory to Applications*, pages 3353–3358, Damascus, Syria, avril 2006.

[31] Xavier Besseron, Christophe Laferrière, Daouda Traoré et Thierry Gautier : X-Kaapi : Une nouvelle implémentation extrême du vol de travail. *In Proceedings des 19èmes rencontres francophones du parallélisme (RenPar'19)*, Toulouse, France, septembre 2009.

[32] Xavier Besseron, Laurent Pigeon, Thierry Gautier et Samir Jafar : Un protocole de sauvegarde/reprise coordonné pour les applications à flot de données reconfigurables. *In Proceedings des 17èmes rencontres francophones du parallélisme (RenPar'17)*, Perpignan, France, octobre 2006.

[33] Xavier Besseron, Laurent Pigeon, Thierry Gautier et Samir Jafar : Un protocole de sauvegarde/reprise coordonné pour les applications à flot de données reconfigurables. *Technique et Science Informatiques*, 2007.

[34] Milind Bhandarkar, Laxmikant V. Kalé, Eric de Sturler et Jay Hoeflinger : Object-based adaptive load balancing for MPI programs. *In Proceedings of*

the International Conference on Computational Science, volume 2074 de *Lecture Notes in Computer Science*, pages 108–117, San Francisco, CA, USA, mai 2001.

[35] Christophe Bidan, Valérie Issarny, Titos Saridakis et Apostolos Zarras : A dynamic reconfiguration service for CORBA. *In Proceedings of the fourth International Conference on Configurable Distributed Systems, 1998.*, pages 35–42, mai 1998.

[36] Toby Bloom : *Dynamic Module Replacement in a Distributed Programming System*. Phd thesis, MIT, 1983.

[37] Toby Bloom et Mark Day : Reconfiguration and module replacement in Argus : theory and practice. *Software Engineering Journal*, 8(2):102–108, Mar 1993.

[38] Manuel Blum, Michael Luby et Ronitt Rubinfeld : Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47:549–595, 1990.

[39] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall et Yuli Zhou : Cilk : An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[40] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri et Anton Selikhov : MPICH-V : Toward a scalable fault tolerant MPI for volatile nodes. *Supercomputing Conference*, 0:29, 2002.

[41] Mohamed Slim Bouguerra, Thierry Gautier, Denis Trystram et Jean-Marc Vincent : A flexible checkpoint/restart model in distributed systems. *In International IEEE conference PPAM '09*. Springer, 2010.

[42] Aurélien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier et Frédéric Magniette : MPICH-V2 : a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. *Supercomputing Conference*, 0:25, 2003.

[43] Aurélien Bouteiller, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier et Franck Cappello : MPICH-V project : a multiprotocol automatic fault tolerant MPI. *The International Journal of High Performance Computing Applications*, 20:319–333, Summer 2006.

[44] Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik et Franck Cappello : Coordinated checkpoint versus message log for fault tolerant MPI. *IEEE International Conference on Cluster Computing*, 0:242, 2003.

[45] Jérémy Buisson : *Adaptation dynamique de programmes et composants parallèles*. Thèse de doctorat, INSA de Rennes, septembre 2006.

[46] Jérémy Buisson, Françoise André et Jean-Louis Pazat : Afpac : Enforcing consistency during the adaptation of a parallel component. *Scalable Computing : Practice and Experience*, 7(3):83–95, septembre 2006.

[47] Sayantan Chakravorty et Laxmikant V. Kalé : A fault tolerant protocol for massively parallel systems. *Parallel and Distributed Processing Symposium, International*, 12:212a, 2004.

[48] Tushar Deepak Chandra et Sam Toueg : Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[49] K. Mani Chandy et Leslie Lamport : Distributed snapshots : determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[50] David Chase et Yossi Lev : Dynamic circular work-stealing deque. *In SPAA '05 : Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM.

[51] Djalel Chefrour : *Plate-forme de composants logiciels pour la coordination des adaptations multiples en environnement dynamique*. Thèse de doctorat, Université Rennes 1, 11 2005.

[52] Djalel Chefrour et Françoise André : Auto-adaptation de composants ACEEL coopérants. *In 3ème Conférence Française sur les Systèmes d'Exploitation (CFSE'3)*, La Colle sur Loup, France, octobre 2003.

[53] Wei Chen, Sam Toueg et Marcos K. Aguilera : On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51:13–32, 2002.

[54] Benoit Claudel, Guillaume Huard et 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.

[55] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein : *Introduction to Algorithms*. MIT Press, Cambridge, MA, second édition, 2001.

[56] Camille Coti, Thomas Hérault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez et Franck Cappello : Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. *Supercomputing Conference*, 0:18, 2006.

[57] Pierre-Charles David : *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. Thèse de doctorat, École des Mines de Nantes and Université de Nantes, juillet 2005.

[58] Pierre-Charles David et Thomas Ledoux : WildCAT : a generic framework for context-aware applications. *In MPAC '05 : Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM.

[59] Pierre-Charles David, Thomas Ledoux, Marc Léger et Thierry Coupaye : FPath and FScript : Language support for navigation and reliable reconfiguration of fractal architectures. *Annales des Télécommunications*, 64:45–63, 2009.

[60] Noël De Palma, Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sylvain Sicard et Christophe Taton : Jade : Un environnement d'administration autonome. *Technique et Science Informatiques*, 2008.

[61] Noël De Palma, Philippe Laumay et Luc Bellissard : Ensuring dynamic reconfiguration consistency. *In 6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop*, pages 18–24, 2001.

[62] Anind K. Dey et Gregory D. Abowd : Towards a Better Understanding of Context and Context-Awareness. *CHI 2000 Workshop on the What, Who, Where, When, and How of Context-Awareness*, 2000.

[63] Stéphane Drapeau : *Un Canevas Adaptable de Services de Duplication*. Thèse de doctorat, Institut National Polytechnique de Grenoble - INPG, juin 2003.

[64] Jason Duell, Paul Hargrove et Eric Roman : The design and implementation of Berkeley Lab's Linux Checkpoint/Restart. Rapport technique, Lawrence Berkeley National Laboratory, décembre 2002.

[65] Elmootazbellah N. Elnozahy : *Manetho : fault tolerance in distributed systems using rollback-recovery and process replication*. Phd thesis, Rice University, Houston, TX, USA, 1993.

[66] Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang et David B. Johnson : A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[67] Elmootazbellah N. Elnozahy et James S. 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, mai 2004.

[68] Brian Ensink et Vikram Adve : Coordinating adaptations in distributed systems. *In ICDCS '04 : Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 446–455, Washington, DC, USA, 2004. IEEE Computer Society.

[69] Brian Ensink, Joel Stanley et Vikram Adve : Program Control Language : a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082 – 1104, 2003.

[70] Aurélien Esnard, Michaël Dussere et Olivier Coulaud : A Time-Coherent Model for the Steering of Parallel Simulations. *In Euro-Par 2004 Parallel Processing*, volume 3149 de *Lecture Notes in Computer Science*, pages 90–97, Pisa, Italy, décembre 2004. Springer.

[71] Graham E. Fagg, Antonin Bukovsky et Jack Dongarra : HARNESS and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1495, 2001.

[72] Peter Feiler et Jun Li : Consistency in dynamic reconfiguration. *In CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems*, page 189, Washington, DC, USA, 1998. IEEE Computer Society.

[73] Michael J. Fischer, Nancy A. Lynch et Michael S. Paterson : Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[74] Ian Foster : Globus Toolkit version 4 : Software for service-oriented systems. *In Proceedings of IFIP International Conference on Network and Parallel Computing*, volume 3779 de *Lecture Notes in Computer Science*, pages 2–13, 2005.

[75] Ian Foster et Carl Kesselman : Globus : A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[76] François Galilée : *Athapascan-1 : Interprétation distribuée du flot de données d'un programme parallèle*. Thèse de doctorat, Institut National Polytechnique de Grenoble, septembre 1999.

[77] François Galilée, Jean-Louis Roch, Gerson Cavalheiro et Matthias Doreille : Athapascan-1 : On-line building data flow graph in a parallel language. *In International Conference on Parallel Architectures and Compilation Techniques, PACT '98*, pages 88–95, Paris, France, octobre 1998.

[78] Thierry Gautier, Xavier Besseron et Laurent Pigeon : Kaapi : a thread scheduling runtime system for data flow computations on cluster of multi-processors. *In Parallel Symbolic Computation 2007 (PASCO '07)*, London, Ontario, Canada, 2007. ACM.

[79] Thierry Gautier, Rémi Revire et Jean-Louis Roch : Athapascan : API for asynchronous parallel programming. Rapport technique, Projet APACHE, INRIA Rhône-Alpes, février 2003.

[80] Thierry Gautier, Jean-Louis Roch et Frédéric Wagner : Fine grain distributed implementation of a dataflow language with provable performances. *In ICCS '07 : Proceedings of the 7th international conference on Computational Science, Part II*, Beijing, China, mai 2007. Springer.

[81] Gabor Gombás, Csaba Attila Marosi et Zoltán Balaton : Grid application monitoring and debugging using the Mercury monitoring system. *In EGC*, pages 193–199, 2005.

[82] Pierre guillaume Raverdy, Hubert Le Van Gong et Rodger Lea : DART : A reflective middleware for adaptive applications. *In OOPSLA '98 : Workshop #13 : Reflective programming in C++ and Java*, 1998.

[83] High Performance Fortran Forum. *High Performance Fortran Language Specification, version 2.0*, janvier 1997.

[84] Christine R. Hofmeister : *Dynamic reconfiguration of distributed applications*. Phd thesis, University of Maryland, College Park, College Park, MD, USA, 1993.

[85] Christine R. Hofmeister et James M. Purtilo : A framework for dynamic reconfiguration of distributed programs. *In Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 560–571, 1993.

[86] Chao Huang : System support for checkpoint and restart of Charm++ and AMPI applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[87] Chao Huang, Gengbin Zheng et Laxmikant V. Kalé : Supporting adaptivity in MPI for dynamic parallel applications. Rapport technique, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[88] Kuang-Hua Huang et Jacob A. Abraham : Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528, 1984.

[89] Eduardo Huedo, Rubén S. Montero et Ignacio M. Llorente : The GridWay framework for adaptive scheduling and execution on grids. *Scalable Computing : Practice and Experience*, 6:1–8, 2005.

[90] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox et 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, 03 2007.

[91] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger et Chung-Yee Lee : Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[92] Samir Jafar : *Programmation des systèmes parallèles distribuées : tolérance aux pannes, résilience et adaptabilité*. Thèse de doctorat, Institut National Polytechnique de Grenoble (INPG), juillet 2006.

[93] Samir Jafar, Axel W. Krings et Thierry Gautier : Flexible rollback recovery in dynamic heterogeneous grid computing. *IEEE Transactions on Dependable and Secure Computing*, 2008.

[94] Samir Jafar, Axel W. Krings, Thierry Gautier et Jean-Louis Roch : Theft-induced checkpointing for reconfigurable dataflow applications. *In EIT '05 : IEEE Electro/Information Technology Conference*, Lincoln, Nebraska, mai 2005.

[95] George Karypis et Vipin Kumar : A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[96] Richard Koo et Sam Toueg : Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, Jan. 1987.

[97] Jeff Kramer et Jeff Magee : Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, avril 1985.

[98] Jeff Kramer et Jeff Magee : The evolving philosophers problem : dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov 1990.

[99] Alexey Kukanov et Michael J. Voss : The foundations for scalable multi-core software in Intel® Threading Building Blocks. *Intel Technology Journal*, novembre 2007.

[100] Ten H. Lai et Tao H. Yang : On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[101] Leslie Lamport : Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[102] Pierre Lemarinier, Aurélien Bouteiller, Thomas Hérault, Géraud Krawezik et Franck Cappello : Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. *In CLUSTER '04 : Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 115–124, Washington, DC, USA, 2004. IEEE Computer Society.

[103] Jean-Denis Lesage et Bruno Raffin : A hierarchical component model for large parallel interactive applications. *Journal of Supercomputing*, 7(1):67–8, juillet 2008.

[104] Michael Litzkow, Todd Tannenbaum, Jim Basney et Miron Livny : Checkpoint and migration of UNIX processes in the Condor distributed processing system. Rapport technique, University of Wisconsin - Madison Computer Sciences Department, avril 1997.

[105] Soulla Louca, Neophytos Neophytou, Adrianos Lachanas et Paraskevas Evripidou : MPI-FT : Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, 10(4):371–382, décembre 2000.

[106] Jason Maassen, Rob V. van Nieuwpoort, Thilo Kielmann, Kees Verstoep et Mathijs den Burger : Middleware adaptation with the Delphoi service. *Concurrency and Computation : Practice and Experience*, 18(13):1659–1679, 2006.

[107] Kaoutar El Maghraoui, Boleslaw K. Szymanski et Carlos Varela : An architecture for reconfigurable iterative MPI applications in dynamic environments. *In Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM'2005)*, volume 3911 de *Lecture Notes in Computer Science*, pages 258–271. Springer, 2005.

[108] Message Passing Interface Forum. *MPI : A Message-Passing Interface Standard*, mai 1994.

[109] Message Passing Interface Forum. *MPI-2 : Extensions to the Message-Passing Interface*, juillet 1997.

[110] Kaveh Moazami-Goudarzi : *Consistency preserving dynamic reconfiguration of distributed systems*. Phd thesis, Imperial College, 1999.

[111] Kaveh Moazami-Goudarzi et Jeff Kramer : Maintaining node consistency in the face of dynamic change. *International Conference on Configurable Distributed Systems*, 0:62, 1996.

[112] Frank Mueller : A library implementation of POSIX threads under UNIX. *In USENIX Conference*, pages 29–41, 1993.

[113] Robert H. B. Netzer et Jian Xu : Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[114] Daniel Paul, Sudhakar Yalamanchili, Karsten Schwan et Rakesh Jha : Decision models for adaptive resource management in multiprocessor systems, 1998.

[115] François Pellegrini et Jean Roman : Scotch : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *In High-Performance Computing and Networking*, pages 493–498, 1996.

[116] Laurent Pigeon : Conception d'une bibliothèque pour les opérations de communication collective pour le langage de haut niveau Athapascan. Dea d'informatique : Systèmes et communications, Université Joseph Fourier, 2003.

[117] Laurent Pigeon : *Environnement interopérable distribué pour les simulations numériques avec composants CAPE-OPEN*. Thèse de doctorat, Institut National Polytechnique de Grenoble (INPG), septembre 2007.

[118] James S. Plank : *Efficient checkpointing on MIMD architectures*. Phd thesis, Princeton University, Princeton, NJ, USA, 1993.

[119] James S. Plank : A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Rapport technique, University of Tennessee, juillet 1996.

[120] James S. Plank, Micah Beck, Gerry Kingsley et Kai Li : Libckpt : Transparent checkpointing under Unix. *In Usenix Winter Technical Conference*, pages 213–223, janvier 1995.

[121] James S. Plank, Scott Simmerman et Catherine D. Schuman : Jerasure : A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Rapport technique, University of Tennessee, août 2008.

[122] James S. Plank, Jian Xu et Robert H. B. Netzer : Compressed differences : An algorithm for fast incremental checkpointing. Rapport technique, University of Tennessee, août 1995.

[123] Brian Randell : System structure for software fault tolerance. *In Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[124] Sriram Rao, Lorenzo Alvisi et Harrick M. Vin : Egida : An extensible toolkit for low-overhead fault-tolerance. 0:48, 1999.

[125] Sriram Rao, Lorenzo Alvisi et Harrick M. Vin : The cost of recovery in message logging protocols. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):160–173, 2000.

[126] Choopan Rattanapoka : *P2P-MPI : A fault-tolerant Message Passing Interface Implementation for Grids*. Thèse de doctorat, University Louis Pasteur, Strasbourg, avril 2008.

[127] Daniel A. Reed et Celso L. Mendes : Intelligent monitoring for adaptation in grid applications. *Proceedings of the IEEE*, 93(2):426–435, Feb. 2005.

[128] Glenn Ricart et Ashok K. Agrawala : An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[129] Michael Rieker, Jason Ansel et Gene Cooperman : Transparent user-level checkpointing for the native Posix thread library for Linux. *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 492–498, 2006.

[130] Martin C. Rinard et Monica S. Lam : The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, 1998.

[131] Thomas Roche : *Dimensionnement et intégration d'un chiffre symétrique dans le contexte d'un système d'information distribué de grande taille*. Thèse de doctorat, Laboratoire d'Informatique de Grenoble, France, 2010.

[132] Seyed M. Sadjadi : A survey of adaptive middleware, 2003.

[133] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove et Eric Roman : The LAM/MPI checkpoint/-restart framework : System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[134] Xuanhua Shi, Jean-Louis Pazat, Eric Rodriguez, Hai Jin et Hongbo Jiang : Adapting grid applications to safety using fault-tolerant methods : Design, implementation and evaluations. *Future Generation Computer Systems*, 26(2):236 – 244, 2010.

[135] Otto Sievert et Henri Casanova : A simple MPI process swapping architecture for iterative applications. *International Journal of High Performance Computing Applications*, 18(3):341–352, 2004.

[136] Georg Stellner : CoCheck : Checkpointing and process migration for MPI. *International Parallel Processing Symposium*, 0:526, 1996.

[137] Robert E. Strom et Shaula Yemini : Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[138] Yuval Tamir et Carlo H. Séquin : Error recovery in multicomputers using global checkpoints. *In Proc. of 1984 International Conference on Parallel Processing*, pages 32–41, août 1984.

[139] Brian Tierney, Ruth A. Aydt, Dan Gunter, Warren Smith, Martin Swany, Valerie E. Taylor et Rich Wolski : A Grid Monitoring Architecture, 2002.

[140] Daouda Traoré : *Algorithmes parallèles auto-adaptatifs et applications*. Thèse de doctorat, Institut Polytechnique de Grenoble (INPG), décembre 2008.

[141] Daouda Traoré, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier et Julien Bernard : Deque-free work-optimal parallel STL algorithms. *In Euro-Par 2008 Parallel Processing*, volume 5168 de *Lecture Notes in Computer Science*, Las Palmas, Spain, août 2008. Springer.

[142] Kishor S. Trivedi : *Probability and statistics with reliability, queuing and computer science applications*. John Wiley and Sons, Chichester, UK, 2001.

[143] Sathish S. Vadhiyar et Jack Dongarra : Self adaptivity in grid computing. *Concurrency and Computation : Practice and Experience*, 17:235–257, 2005.

[144] Sathish S. Vadhiyar, Graham E. Fagg et Jack Dongarra : Automatically tuned collective communications. *In 2000 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[145] Leslie G. Valiant : A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[146] Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann et Henri E. Bal : Satin : Simple and efficient java-based grid programming. *Scalable Computing : Practice and Experience*, 6(3):19–32, septembre 2005.

[147] Marco Vanneschi et Luca Veraldi : Dynamicity in distributed applications : issues, problems and the ASSIST approach. *Parallel Computing*, 33(12):822 – 845, 2007.

[148] Manjunath G. Venkata et Patrick G. Bridges : MPI/CTP : A reconfigurable MPI for HPC applications. *In PVM/MPI*, pages 96–104, 2006.

[149] Manjunath G. Venkata, Patrick G. Bridges et Patrick M. Widener : Using application communication characteristics to drive dynamic MPI reconfiguration. *In IPDPS '09 : Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.

[150] John Wakerly : *Error detecting codes, self-checking circuits and applications*. Computer Design and Architecture Series. Elsevier, 1978.

[151] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin et W. Kent Fuchs : Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546–554, 1995.

[152] Hal WASSERMAN et Manuel BLUM : Software reliability via run-time result-checking. *Journal of the ACM*, 44:826–849, 1994.

[153] Michel WERMELINGER : Towards a chemical model for software architecture reconfiguration. *In CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems*, pages 111–118. IEEE Computer Society, 1998.

[154] Michel WERMELINGER et José L. FIADEIRO : A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133 – 155, 2002.

[155] Matthias WIESMANN, Fernando PEDONE et André SCHIPER : A systematic classification of replicated database protocols based on atomic broadcast. *In ERSADS '99 : 3rd Europeean Research Seminar on Advances in Distributed Systems*, 1999.

[156] Rich WOLSKI, Neil T. SPRING et Jim HAYES : The Network Weather Service : a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15:757–768, 1999.

[157] Gosia WRZESINSKA : *Handling complexity and change in grid computing.* Phd thesis, Vrije Universiteit, Amsterdam, The Netherlands, mai 2007.

[158] Gosia WRZESINSKA, Jason MAASSEN et Henri E. BAL : Self-adaptive applications on the grid. *In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*, San Jose, CA, USA, mars 2007.

[159] Gosia WRZESINSKA, Rob V. van NIEUWPOORT, Jason MAASSEN, Thilo KIELMANN et Henri E. BAL : Fault-tolerant scheduling of fine-grained tasks in grid environments. *International Journal of High Performance Computing Applications*, 20(1):103–114, 2006.

[160] Tao YANG : *Scheduling and code generation for parallel architectures.* Phd thesis, Rutgers University, New Brunswick, NJ, USA, 1993.

[161] Tao YANG et Apostolos GERASOULIS : DSC : Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[162] Andrew J. YOUNG et Jeff MAGEE : A flexible approach to evolution of reconfigurable systems. *In International Workshop on Configurable Distributed Systems, 1992*, pages 152–163, Mar 1992.

[163] John W. YOUNG : A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[164] Gengbin ZHENG, Lixia SHI et Laxmikant V. KALÉ : FTC-Charm++ : an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. *IEEE International Conference on Cluster Computing*, 0:93–103, 2004.

Résumé

Tolérance aux fautes et reconfiguration dynamique pour les applications distribuées à grande échelle

Ce travail se place dans le cadre du calcul haute performance sur des plateformes d'exécution de grande taille telles que les grilles de calcul. Les grilles de calcul sont notamment caractérisées par (1) des changements fréquents des conditions d'exécution et, en particulier, par (2) une probabilité importante de défaillance due au grand nombre de composants. Pour exécuter une application efficacement dans un tel environnement, il est nécessaire de prendre en compte ces paramètres.

Nos travaux de recherche reposent sur la représentation abstraite de l'application sous forme d'un graphe de flot de données de l'environnement de programmation parallèle et distribuée Athapascan/Kaapi. Nous utilisons cette représentation abstraite pour apporter des solutions aux problèmes (1) de reconfiguration dynamique et (2) de tolérance aux fautes.

- Tout d'abord, nous proposons un mécanisme de reconfiguration dynamique qui gère, de manière transparente pour le programmeur de la reconfiguration, les problèmes d'accès concurrents sur l'état de l'application et la cohérence mutuelle des états en cas de reconfiguration distribuée.
- Ensuite, nous présentons un protocole de tolérance aux fautes original qui permet d'effectuer une reprise partielle de l'application en cas de panne. Pour cela, il détermine l'ensemble des tâches de calcul strictement nécessaires à la reprise de l'application.

Ces contributions sont évaluées en utilisant les logiciels Kaapi et X-Kaapi sur la plateforme de calcul Grid'5000.

Mots-clés : Calcul parallèle, Grille de calcul, Adaptation et reconfiguration dynamique, Tolérance aux fautes, Graphe de flot de données.

Abstract

Fault tolerance and dynamic reconfiguration for large scale distributed applications

This work deals with high performance computing on large scale platforms like computing grids. Computing grids are characterized by (1) frequent changes in execution context and, especially, by (2) a high failure probability caused by the large number of components. Running an application efficiently in such an environment requires to consider these parameters.

Our research work is based on the abstract representation of the application as a data flow graph from the parallel and distributed programming model Athapascan/Kaapi. This abstract representation is used to provide solutions for (1) dynamic reconfiguration and (2) fault tolerance issues.

- First, we propose a dynamic reconfiguration mechanism that manages, transparently for the reconfiguration programmer, concurrent operations on the application state and mutual consistency of states for distributed reconfiguration.
- Secondly, we present an original fault tolerance protocol that allows partial rollback of the application in case of failure. For this purpose, the set of strictly required computation tasks to recover is computed.

These contributions are evaluated through the Kaapi and X-Kaapi software on the Grid'5000 computing platform.

Keywords : Parallel computing, Grid computing, Dynamic adaptation and reconfiguration, Fault tolerance, Data flow graph.