Université Joseph Fourier

U.F.R Informatique & Mathématiques Appliquées

Institut National Polytechnique de Grenoble

ENSIMAG

I.M.A.G.

ÉCOLE DOCTORALE
MATHÉMATIQUES ET INFORMATIQUE

MASTER 2 RECHERCHE :
SYSTÈMES ET LOGICIELS

Projet présenté par :

Xavier Besseron

CCK : un protocole coordonné de sauvegarde/reprise pour la tolérance aux pannes des applications itératives en calcul numérique

Effectué au laboratoire ID-imag

Laboratoire
Informatique et
Distribution

Date : Lundi 19 Juin 2006
11 h 00 – Salle F116

Jury :	J. Coutaz
	J.-F. Méhaut
	R. Echahed
Examinateur :	N. De Palma
Encadrant :	T. Gautier

Résumé

Malgré l'augmentation permanente des performances des ordinateurs, les applications de calcul scientifique demandent toujours plus de puissance. Les ingénieurs et les chercheurs ont proposé d'utiliser des architectures distribuées qui regroupent plusieurs centaines voir milliers d'ordinateurs interconnectés par des réseaux rapides. On parle alors de grille de calcul. Ces architectures sont caractérisées par une forte hétérogénéité de leurs composants (processeur, système, réseau) et une disponibilité très variable. Du fait du grand nombre de composants non fiables, la probabilité d'apparition d'une panne très forte, en particulier pour les applications qui nécessitent de long temps d'exécution sur beaucoup de processeurs.

L'exploitation de ces architectures nécessite l'utilisation de mécanismes pour se prémunir des conséquences de la défaillance de certains leurs composants.

Ce travail a pour objectif de concevoir un protocole de tolérance aux pannes pour des applications itératives de calcul scientifique qui soit, d'une part, efficace et, d'autre part, transparent pour l'utilisateur. Le protocole CCK (Coordinated Checkpointing in KAAPI) est conçu à partir d'un protocole standard de sauvegarde/reprise coordonnée auquel on apporte plusieurs améliorations. Ces améliorations portent à la fois sur l'étape de coordination de la sauvegarde et sur la phase de redémarrage. Elles reposent sur une représentation abstraite du futur de l'exécution de l'application sous forme d'un graphe de flot de données.

Le protocole CCK est entièrement détaillé : l'algorithme est présenté, sa correction est prouvée et son coût est analysé. Des expérimentations préliminaires mettent en évidence les paramètres importants liés à l'étape de sauvegarde de ce protocole sur une applications typiques. Des simulations évaluent le coût d'un redémarrage. Enfin, plusieurs optimisations et perspectives sont proposées.

Remerciements

Je voudrais tout d'abord remercier mon tuteur Thierry Gautier que je connais depuis près d'un an. Toujours à l'écoute, il donne toujours de bons conseils et il m'a appris/permis d'apprendre énormément de choses.

Je remercie Samir Jafar qui a été mon guide pour ce stage dans l'univers des protocoles de tolérance aux pannes ; je te souhaite une bonne continuation pour ton retour en Syrie.

Je remercie également Laurent Pigeon, mon collègue de bureau (un supporter brésilien à ses heures perdues) avec qui j'ai fait quelques séances de déboguage téléphonique lorsque plus rien ne marchait. (Au fait, il avance ton premier chapitre ?). C'est un vrai plaisir de travailler avec l'équipe des KAAPIstes.

Merci aussi à Jean-Denis, Johan, Maxime, tous les stagiaires du bout du couloir et à tous les gens du laboratoire ID pour leur accueil. Enfin je remercie mes amis, ma famille et ceux qui ont relu mon rapport.

Table des matières

Chapitre 1

Introduction

Durant ces dernières années, la puissance des ordinateurs n'a cessé de croître. Cependant les applications de calcul scientifique demandent de plus en plus de ressources. Malgré l'augmentation de la puissance des processeurs, on a été amené à construire des systèmes informatiques de plus en plus grands, notamment en utilisant plusieurs processeurs pour exploiter le parallélisme des applications.

Ainsi, on est passé des simples architectures SMP (Symetric Multi-Processors) regroupant plusieurs processeurs utilisant une mémoire partagée à des grappes qui sont un ensemble de machines (le plus souvent SMP) reliées par un réseau local. La généralisation de ceci amène à considérer des environnements de type grille qui sont la connexion de plusieurs grappes au niveau national ou international. On obtient ainsi des systèmes de calcul répartis pouvant facilement comporter plusieurs milliers de processeurs et plusieurs tera-octets de mémoire vive. De nombreuses applications très gourmandes en temps de calcul et en mémoire cherchent à exploiter au mieux ce type de système.

Cependant, de tels systèmes restent difficiles à exploiter, en particulier à cause du risque important de panne d'un de ses nombreux éléments. La probabilité d'apparition d'une panne augmente avec le nombre de composants du système, il est donc très probable qu'une panne interviennent durant une exécution longue d'une application. Un axe de recherche important vise à permettre à ce type d'applications de tolérer les pannes, de manière matérielle ou logicielle, transparente ou non.

1.1 Contexte et objectifs

Les travaux présentés dans ce rapport portent essentiellement sur des applications parallèles s'exécutant sur des environnements de type grappe ou grille de calcul. Les applications parallèles visées sont des applications itératives de calcul scientifique qui simulent itérativement l'évolution des phénomènes physiques. On peut ainsi citer des applications de dynamique moléculaire [21], de réalité virtuelle [37] ou de décomposition de domaine [48, 13]. Ces applications sont caractérisées par un temps d'exécution long et par un schéma de communication qui se reproduit d'itération en itération. L'objectif est de concevoir un protocole de tolérance aux pannes adapté à ce type d'applications qui soit efficace (c'est-à-dire qui ne perturbe pas ou peu les performances). De plus, ce dispositif doit être transparent pour le programmeur et l'utilisateur de l'application. Une contrainte supplémentaire est de permettre son utilisation dans un contexte hétérogène (architectures et systèmes d'exploitation variés).

Nous baserons notre travail à un niveau intermédiaire entre le système et les applications. De nombreux travaux ont montré la possibilité d'offrir des protocoles pour la tolérance aux pannes au niveau système, mais les solutions proposées souffrent d'un manque d'efficacité et de portabilité sur différents systèmes et architectures. Nous nous baserons sur l'intergiciel KAAPI qui permet d'ordonnancer des applications parallèles sur une grille de calcul. L'intérêt de cet intergiciel par rapport aux autres est sa capacité à construire une représentation abstraite de l'exécution future du calcul sous la forme d'un graphe de flot de données. Ce graphe sert actuellement au calcul d'un ordonnancement. Nous l'utiliserons pour proposer un algorithme efficace de tolérance aux pannes.

1.2 Contributions

Le problème de la tolérance aux pannes dans les applications parallèles et les systèmes répartis n'est pas un problème nouveau. Le chapitre 2 présente un récapitulatif des différentes approches qui ont été réalisées pour aborder ce problème. Une grande difficulté est d'arriver à conserver, capturer ou reconstituer un état global cohérent. Ce chapitre décrit plus particulièrement les techniques basées sur l'utilisation d'une mémoire stable qui sont plus adaptées aux applications de calculs parallèles. Parmi ces techniques, on distingue deux types d'approches : les approches par journalisation d'événements et les approches par sauvegarde de l'état des processus.

Les premières sauvegardent sur une mémoire stable tous les événements non déterministes qui modifient l'état de l'application. Ces événements non déterministes correspondent en général à la réception de certains messages. De telles approches peuvent avoir un surcoût important à l'exécution, en particulier avec les applications qui communiquent beaucoup. En cas de panne, l'état global cohérent de l'application est recréé en rejouant les événements sauvegardés.

Les approches par sauvegarde de l'état des processus effectuent régulièrement et/ou sous certaines conditions une sauvegarde de l'état des processus. Le surcoût est plus faible à l'exécution. La construction de l'état global cohérent peut se faire à la sauvegarde (en coordonnant les processus) ou à la reprise en effectuant des calculs de dépendance.

Le chapitre 3 détaille et analyse un protocole de sauvegarde coordonnée et de reprise globale. Ce protocole a servi de base à ce stage car il est bien adapté au domaine d'applications considéré. Je me suis posé trois questions fondamentales relatives à ce protocole standard :

1. Comment réduire le coût de la coordination lors de la sauvegarde ?
2. Comment accélérer le redémarrage des processeurs après une défaillance de l'un d'entre eux ?
3. Comment réduire la quantité de temps d'exécution perdu en cas de panne ?

J'apporte une réponse à (1) en proposant un algorithme de coordination inspiré de [24] qui utilise les dépendances entre les processus pour réduire le nombre de messages entre les processus. La réponse à (2) est réalisée en utilisant une méthode proposée dans [15, 10] qui vise à conserver une copie locale de la sauvegarde (en plus de la copie sur mémoire stable). Enfin je propose une solution originale pour (3). Cette solution consiste à calculer un ensemble minimal des calculs à ré-exécuter pour garantir la cohérence de l'état global tout en conservant le bénéfice des calculs effectués depuis la dernière sauvegarde. De plus cette solution permet un redémarrage quasi-partiel des processus de l'application à la différence des algorithmes classiques qui requièrent un re-démarrage total de tous les processus : avec notre algorithme, la charge de calcul induite par le redémarrage peut être redistribuée sur l'ensemble des processus non défaillants.

Les solutions (1) et (3) sont possibles car on tire parti de la représentation abstraite de notre application. Cette représentation abstraite est un modèle graphe de flot de données qui permet de connaître le futur de l'exécution. Toutes ces solutions réunies ont permis la conception du protocole CCK (Coordinated Checkpoiting in KAAPI). Je décris tout d'abord formellement ce protocole : le détail des algorithmes, les preuves et les analyses de coût sont présentés au chapitre 4. Des optimisations supplémentaires sont également envisagées. Je l'ai aussi partiellement implémenté et j'ai réalisé des expérimentations pour évaluer ses performances. Les résultats de ces expériences sont présentés au chapitre 5.

Enfin dans le chapitre 6, je présente plusieurs perspectives d'évolution et d'amélioration pour le protocole CCK et je réalise un bilan des travaux effectués au cours de ce stage.

Ce travail a fait l'objet de la rédaction de deux articles dont un accepté :

- Xavier Besseron, Samir Jafar, Thierry Gautier et Jean-Louis Roch. CCK : An improved coordinated checkpoint/rollback protocol for data flow applications in KAAPI. In *Proceedings of the ICTTA'06 2nd IEEE International Conference On Information & Communication Technologies : From Theory To Applications*, Damas, Syria, April 2006.
- Xavier Besseron, Laurent Pigeon, Thierry Gautier, Samir Jafar. Un protocole de sauvegarde / reprise coordonné pour les applications à flot de données reconfigurables. Soumis à *RenPar'17*.

1.3 Cadre de travail

Ce stage a été effectué au laboratoire ID-IMAG, au sein du projet MOAIS (Multi-programmation et Ordonnancement sur ressources distribuées pour les Applications Interactives de Simulation) [30], un projet commun CNRS, INRIA, INPG, UJF. MOAIS étudie la programmation des applications qui s'adaptent à une variation du nombre de ressources tout en cherchant à garantir des performances. La construction d'algorithmes parallèles répartis adaptatifs est l'un des axes de recherche de MOAIS dans lequel s'inscrit ce travail.

Chapitre 2

État de l'art

Ce chapitre définit tout d'abord la notion de panne et de tolérance aux pannes puis présente les différentes techniques de tolérance aux pannes. Nous détaillerons davantage les méthodes de tolérance aux pannes par reprise qui sont au cœur de notre travail. La présentation d'un protocole coordonné standard de sauvegarde/reprise auquel nous nous comparerons fera l'objet du chapitre suivant.

2.1 Système réparti

Un système réparti est un ensemble de processus qui coopèrent et qui s'exécutent sur des machines distinctes. Ces machines sont connectées par un réseau d'interconnexion. Les processus communiquent par des messages en utilisant des canaux de communications. Les canaux de communications peuvent être considérés FIFO (First In, First Out), fiables ou non selon le modèle utilisé.

De nombreux caractéristiques sont liés à ce type de système :

– Il n'y a pas de mémoire commune.
– Il n'y a pas d'horloge commune.
– Les communications sont asynchrones.
– Les sites d'exécution peuvent être hétérogènes.

Ces caractéristiques expliquent les problèmes suivant. Tout d'abord, l'état de l'application est réparti parmi tous les processus. De plus, on ne peut donc pas toujours définir un ordre sur les événements. Enfin, on ne connaît pas de borne supérieure au temps de transmission d'un message.

2.2 Non fiabilité dans les systèmes répartis de grande taille

Pour les applications considérées comme les simulations de systèmes physiques, la quantité de calculs à effectuer peut être très importante. Ainsi l'exécution de certaines instances peut durer plusieurs semaines sur 1000 processeurs.

Cependant, les environnements de type grappe ou grille qui permettent d'exécuter de telles applications ont un risque important de voir un de leurs composants tomber en panne. En effet, la probabilité qu'une panne se produise durant l'exécution augmente de manière considérable avec le nombre de processeurs utilisés [44].

[39] propose une modélisation dans laquelle les pannes d'une machine apparaissent de manière indépendante avec un taux constant $\lambda = \frac{1}{MTBF}$ ($MTBF$, *Mean Time Between Failures*, est le temps moyen entre deux pannes). On définit la fiabilité $R(t)$ d'un système comme la probabilité qu'il soit opérationnel à tout instant $s \in [0, t]$. Pour un système comportant n machines, on a alors $R(t) = e^{-\lambda tn}$. De même, pour la non fiabilité $F(t)$:

$$F(t) = 1 - R(t) = 1 - e^{-\lambda tn}$$

La figure 2.1 montre que la probabilité d'une défaillance augmente considérablement en fonction du nombre de processeurs utilisés et du temps de calcul. Pour le domaine d'application que nous considé-

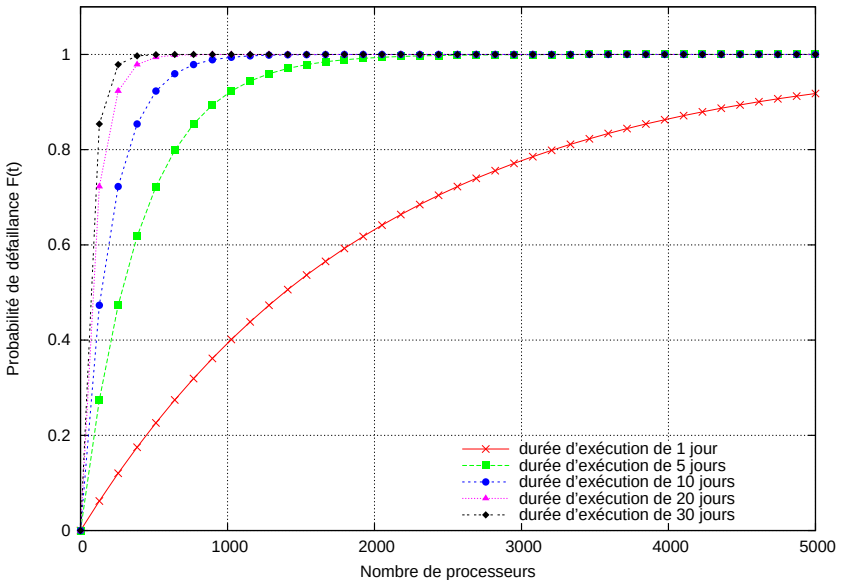


FIG. 2.1 – Probabilité de défaillance dans les environnements parallèles répartis pour différentes durées d'exécution, avec $MTBF$ = 2000 jours $\tilde{6}$ ans pour chaque processeur (i.e. $\lambda = 0.0005$)

rons, on peut avoir des calculs d'une semaine sur 500 processeurs, soit $F(7jours) = 1 - e^{-\lambda \times 7 \times 500} = 0.83$ avec $MTBF$ = 2000 jours. On a donc une probabilité de 83% d'avoir une panne durant un tel calcul.

2.3 Tolérance aux pannes

La tolérance aux pannes est un moyen destiné à assurer la sûreté de fonctionnement d'un système informatique et ainsi de permettre à l'utilisateur de faire confiance au service qui lui est délivré [28]. Durant le fonctionnement d'un système, une faute ou une panne peut survenir. La faute peut entraîner une erreur. Si l'erreur se propage, elle peut alors provoquer une défaillance du système, c'est-à-dire que le service rendu par le système ne correspond plus au service spécifié [3].

Différentes approches sont alors possibles pour permettre à un système de fournir un service auquel l'utilisateur peut avoir confiance [28]. Les plus connues sont les suivantes :

1. **La prévention des fautes** qui tente éviter l'apparition de fautes durant l'exécution. Ceci est généralement réalisé par la vérification des modèles conceptuels.
2. **L'élimination des fautes** qui essaye de réduire le nombre de fautes et leur gravité. On utilise pour cela des méthodes statistiques de preuve de la validité du système (simulations, preuves analytiques, tests, *etc.*).
3. **La prévision des fautes** qui évalue le nombre de fautes et leurs conséquences. L'approche utilisée est alors l'injection de fautes pour valider le comportement du système relativement à ces fautes.
4. **La tolérance aux pannes** (ou aux fautes) qui essaye de masquer l'occurrence des fautes et de fournir le service spécifié malgré leur apparition.

Dans la suite de ce rapport, nous allons uniquement nous intéresser aux méthodes de tolérances aux pannes.

2.3.1 Pannes

Les pannes pouvant survenir durant l'exécution d'applications parallèles peuvent être très variées. Les classements proposés pour distinguer les différents types de pannes s'appuient sur différents critères. Les principaux critères utilisés dans la littérature [3] sont les suivants.

Classement selon la gravité. On distingue alors les pannes franches (soit le système fonctionne, soit il ne fait rien), les pannes par omission (des messages sont perdus), les pannes de temporisation (le temps de réaction du système est en dehors des plages spécifiées), les pannes par valeurs (les résultats fournis sont incorrects) et les pannes byzantines (le système réagit de manière totalement aléatoire).

Classement selon la persistance temporelle. On trouve les pannes transitoires (isolées dans le temps), les pannes intermittentes (aléatoires et répétées) et les pannes permanentes (définitives jusqu'à réparation).

Classement selon la nature. On différencie les pannes accidentelles et intentionelles.

Dans la suite, on ne traitera que des pannes franches ou des pannes qui peuvent être modélisées ou considérées comme des pannes franches. Par exemple, on peut imaginer que si un processus émet un message erroné ou s'il dépasse une temporisation, il est exclu du calcul et considérer qu'il a subi une panne franche. De même, on peut traiter le cas de la volatilité des machines dans un environnement dynamique en considérant la perte d'une ressource comme une panne franche.

2.3.2 Détection des pannes

La tolérance aux pannes nécessite un dispositif de détection de pannes. Cette fonction est réalisée par un détecteur de panne qui permet d'indiquer quand survient une erreur et quelle partie du système est affectée par cette erreur. De nombreux travaux [11] portent sur les mécanismes de détection d'erreurs mais ils sont en dehors du cadre de cette étude. Dans le reste du rapport, on supposera l'existence d'un tel détecteur.

2.4 Techniques de tolérance aux pannes

La tolérance aux pannes apparaît comme un élément indispensable aux systèmes repartis. Pour répondre à ce besoin, plusieurs techniques ont été conçues. Elles reposent toutes sur un mécanisme de redondance [2]. Cette redondance peut être spatiale (duplication 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 basés sur la duplication et les mécanismes basés sur une mémoire stable. Nous présenterons dans les paragraphes suivants ces mécanismes.

2.4.1 Duplication

La tolérance aux pannes par duplication consiste à utiliser des copies multiples d'un même composant ou processus. De cette manière, en cas de défaillance d'un des composants, la panne 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 trois stratégies principales [31] permettant d'assurer cette cohérence :

La duplication active. Ce terme est utilisé pour les stratégies dans laquelle toutes les copies jouent une rôle identique. Les mêmes opérations sont effectuées sur toutes les copies.

La duplication passive pour laquelle on distingue une copie primaire et les copies secondaires. La copie primaire est la seule à effectuer toutes les opérations et elle est remplacée par une copie secondaire en cas de panne.

La duplication semi-active est une combinaison des deux stratégies précédentes : on distingue toujours la copie primaire et les copies secondaires mais les copies secondaires effectuent également les traitements.

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

2.4.2 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 [14]. Une mémoire stable doit préserver l'intégrité des données et les garder accessibles, même en cas de pannes.

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

- Pour un système qui ne tolère qu'une seule panne (respectivement n pannes), la mémoire stable peut être réalisée par la mémoire volatile d'un autre processus (respectivement de n autres processus).
- Dans un système qui ne souhaite tolérer que les pannes transitoires, la mémoire stable peut correspondre au disque dur local du processus.
- Pour un système qui veut tolérer un nombre quelconque de pannes permanentes, la mémoire stable doit être réalisée sur un nœud extérieur aux nœuds de calcul et qui est supposé être fiable.

Le principe de la tolérance aux pannes par mémoire stable est, en cas de panne, de rétablir l'application dans un état cohérent en utilisant les informations stockées sur la mémoire stable. Une fois la panne détectée, un mécanisme de recouvrement se met en place. Deux techniques sont possibles [28] :

La reprise : on remet l'application dans un état correct antérieur grâce à des sauvegardes régulières réalisées en mémoire stable.

La poursuite : on reconstitue un état correct pour l'application sans revenir en arrière. Le recouvrement est en général partiel, on a donc un service dégradé.

Par la suite, nous nous intéresserons uniquement aux techniques par reprise. En effet, un recouvrement partiel n'est pas acceptable pour des applications de calcul scientifique.

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 surcoût qui va dépendre des contraintes imposées au système : nombre et types de pannes à tolérer, reprise globale du système ou uniquement des processus défaillants, etc.

Deux approches sont possibles pour construire un état global cohérent suivant le moment de sa construction [14, 20].

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

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

Dans la suite de ce rapport, nous étudierons les protocoles basés sur une mémoire stable utilisant des techniques de type reprise.

2.5 Tolérance aux pannes par reprise

La tolérance aux pannes par reprise utilise la redondance d'informations. Ces informations correspondent à des sauvegardes de données réalisées aux cours de l'exécution de l'application. Pour cela, la tolérance aux pannes par reprise nécessite une mémoire stable pour stocker les données. De plus, un protocole de tolérance aux pannes par reprise est constitué de deux aspects : la construction d'état global cohérent et la reprise. La reprise peut s'effectuer soit par journalisation, soit par sauvegarde.

2.5.1 État global cohérent

L'état global d'une application parallèle est composé

- de l'état local de tous les processus participants au calcul,
- et de l'état de tous les canaux de communications entre les processus.

Définition 1. *Un **état global cohérent** est un état qui peut se produire durant une exécution correcte 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 envoyé ce message « contient*[1] *» l'envoi de ce message [14, 20].*

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

Les messages orphelins sont la cause de l'incohérence d'un état global car une partie de l'état global est alors reflété par l'état des canaux de communications. Lorsque qu'une panne survient, l'état des canaux de communication est perdu, ce qui provoquent l'apparition de messages orphelins et donc d'incohérences. Le rôle du protocole de tolérance aux pannes par reprise 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 panne, il suffit qu'il soit un état de l'application qui aurait pu se produire durant une exécution sans panne.

Dans la figure 2.2 à droite, l'état global formé des rectangles sur chacune des lignes de temps des trois processus P_0, P_1 etP_2 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'envoie du message m_1. Le message m_1 est un message orphelin.

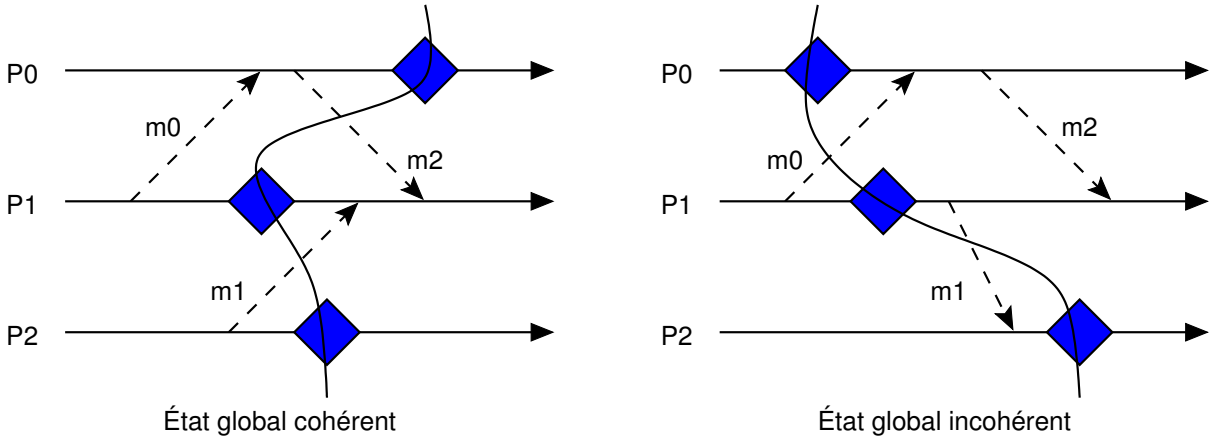


FIG. 2.2 – Exemple d'état global cohérent et incohérent. Les axes horizontaux représentent le temps pour trois processus P_0, P_1 et P_2. Les flèches représentent des communications entre les processus. Les losanges représentent les événements associés à la capture de l'état des processus.

La reprise après panne nécessite de rejouer certaines instructions afin de retrouver l'un des états possibles d'une exécution sans panne. Les instructions qui dépendent d'événement non déterministe, comme la réception de message, peuvent alors modifier l'état d'un processus. La définition suivante modélise l'exécution des processus en tenant compte de ces événements non déterministe.

Définition 3. ***Hypothèse PWD** (PieceWise Deterministic assumption) [42]*

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

Ainsi, si deux processus démarrent du même état et s'ils sont soumis aux mêmes événements non déterministes, ils auront la même exécution et produiront les mêmes sorties. Cela sera donc encore valide pour les processus avant panne et ceux qui reprendrons une exécution à partir d'un état global cohérent.

La définition 2 peut être généralisée aux événements non déterministe quelconques (par exemple des décisions d'ordonnancement).

[1] signifie que l'événement associé à l'envoi précède l'événement associé à la prise d'état du processus.

Définition 4. *On appelle **processus orphelin** un processus dont l'état dépend d'un événement non déterministe qui ne peut être reproduit à la reprise. Ceci peut être causé par un message orphelin.*

L'existence de ces processus orphelins caractérise les états globaux «pathologiques» pour lesquels il n'est pas possible de reprendre une exécution.

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 [14]. La section suivante présente les protocoles de reprise basés sur la journalisation des événements non déterministes. Puis nous présenterons les mécanismes basés sur la reprise par sauvegarde.

2.5.2 Reprise par journalisation

Le principe de la tolérance aux pannes par journalisation est de sauvegarder l'histoire de l'application. Ceci repose sur l'hypothèse PWD. Chaque processus crée un journal des événements 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 son état initial et en rejouant les événements non déterministes inscrits dans son journal. L'hypothèse PWD garantit que cette reconstruction aboutira au même état.

De plus, pour éviter la reconstruction à partir de l'état initial, le processus peut effectuer des sauvegardes périodiques de son état. Il faut également remarquer que ces informations (le journal et les sauvegardes périodiques) doivent être stockées sur une mémoire stable.

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

Journalisation pessimiste

Le protocole de reprise par journalisation pessimiste [42] repose sur l'hypothèse (pessimiste) qu'une panne peut se produire immédiatement après un événement non déterministe. Le principe de ce protocole est donc de ne pas permettre à un processus de dépendre d'un événement non déterministe tant que celui-ci n'a pas été stocké sur un support stable.

Concrètement, si on considère que les événements non déterministes sont uniquement les réceptions de messages, ce protocole impose aux processus de sauvegarder tous les messages reçus avant d'envoyer un message à un autre processus [25]. 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 surcoût conséquent lors d'une exécution sans panne, en particulier pour les applications effectuant beaucoup de communications.

Journalisation optimiste

La journalisation optimiste [42] veut améliorer les performances en faisant l'hypothèse (optimiste) qu'une panne ne se produira pas avant la sauvegarde de l'événement 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 la « condition de non orphelinité ». Par conséquent, lors de la reprise, il peut être nécessaire de revenir plusieurs fois en arrière pour obtenir un état global cohérent. De plus, le calcul pour obtenir l'état global cohérent à la reprise peut avoir un coût important.

Journalisation causale

La journalisation causale [32, 14] combine les avantages de la journalisation pessimiste pour la reprise et les avantages de la journalisation optimiste en ce qui concerne le surcoût 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 [27]) avec les autres événements non déterministes [25]. Ces informations (appelées également le déterminant) sont aussi ajoutées à tous les messages envoyés aux 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 connaît l'historique des événements qui ont produit son état et celui des autres processus. Ces informations le protègent des pannes des autres processus et permettent de garantir la condition de « non orphelinité ».

2.5.3 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 [14].

Ces protocoles n'ont pas besoin de l'hypothèse PWD et donc ils ne nécessitent pas de stocker les événements non déterministes. Il ne garantissent pas de retrouver l'état de l'application précédent la panne, mais uniquement un état global cohérent qui aurait pu se produire durant une exécution normale. En outre, ces protocoles sont moins restrictifs et plus simples à implanter.

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.

Sauvegarde non coordonnée

La sauvegarde non coordonnée [14] 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 [47]. 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** [36] : lors de la construction de l'état global cohérent, les dépendances entre les messages peuvent entraîner 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 surcoût mais ne contribuent pas au redémarrage. Enfin, cette méthode oblige les processus à conserver plusieurs sauvegardes.

Sauvegarde coordonnée

Le principe de la sauvegarde coordonnée est de réaliser un ensemble de sauvegardes des états des processus qui soit cohérent. Cette méthode est basée sur les travaux de Chandy et Lamport [20].

La sauvegarde coordonnée est réalisée par une étape de synchronisation des processus pendant laquelle les calculs sont arrêtés. Cette étape permet d'assurer que l'ensemble des états des processus forme un état global cohérent. Une explication détaillée d'un protocole standard de sauvegarde coordonnée, inspirée de [43], est proposée au chapitre suivant.

L'avantage de cette approche 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 surcoût de stockage. Le principal inconvénient est le surcoût induit par la synchronisation forte des processus. Plusieurs méthodes ont été proposées pour tenter d'améliorer les performances :

– La sauvegarde coordonnée **non bloquante** [20] crée un clone du processus à sauvegarder. Ce clone n'est pas influencé par les nouveaux messages reçus et peut sauvegarder son état tandis que le processus original continue l'exécution normale. pour éviter son blocage.
– La sauvegarde avec **horloges synchronisées** [26] 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 panne), on est assuré de la cohérence des sauvegardes sans avoir échangé de messages.
– La sauvegarde coordonnée **minimale** [24] 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 envoie des messages qui sont diffusés de proche en proche selon les dépendances entre processus) ; durant la deuxième, les processus identifiés réalisent leur sauvegarde.

Sauvegarde induite par les communications

La sauvegarde induite par les communications (Communication-Induced Checkpointing CIC) [5] 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 [33]. Ces notions permettent de déterminer si une sauvegarde sera utile ou non en cas de panne et de reprise.

2.6 Comparaison des protocoles de tolérance aux pannes par reprise

Comme l'a montré la section précédente, il existe plusieurs protocoles de tolérance aux pannes par reprise basés sur l'utilisation d'une mémoire stable. Cette section propose de comparer les différentes méthodes présentées. Ensuite, quelques implémentations de ces protocoles sont citées. On détaille et on compare également les principes de quelques unes de ces implémentations.

2.6.1 Comparaison des protocoles

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

Hypothèse PWD : indique si l'hypothèse PWD est faite par cette technique.

Effet domino : indique s'il y a un risque d'effet domino.

Point de reprise par processus : donne le nombre de points de reprise à conserver.

Processus orphelin : indique s'il y a un risque d'apparition de processus orphelin.

Propagation de la reprise : indique à partir de quel point de reprise la reprise est effectuée.

2.6.2 Implémentations existantes

Plusieurs environnements d'exécution d'applications parallèles ont été rendus tolérants aux pannes en utilisant des techniques de sauvegarde/reprise. On peut citer ProActive [6], Cilk-NOW [7], Satin [46]. La suite en détaille d'autres :

Cocheck [41] est une extension de Condor [29] pour les applications parallèles. Il construit un état global cohérent par une sauvegarde coordonnée. Pour cela, il utilise la bibliothèque Condor. La sauvegarde effectuée par la bibliothèque Condor correspond à l'espace d'adressage du processus ce qui nécessite donc une ressource homogène le redémarrage. De plus, cette sauvegarde ne supporte pas les applications multithreadées.

MPICH-V est une implémentation de MPI qui propose trois variantes pour la tolérance aux pannes. MPICH-V1 [8] est basé sur la sauvegarde non coordonnée avec journalisation des messages. MPICH-causal [9] implante une journalisation causale des événements non déterministes (messages). MPICH-cl [10] réalise une sauvegarde coordonnée où chaque processus stocke sa sauvegarde en local et sur une mémoire stable pour améliorer les performances de la reprise globale. Ces trois versions reposent sur la bibliothèque Condor et donc ne supportent ni l'hétérogénéité, ni le multithreadé.

Charm++ [22] est un langage de programmation parallèle qui repose sur la notion de **chares**. Ce sont des objets concurrents qui peuvent être appelés à distance. Charm++ propose un mécanisme de tolérance aux pannes par une sauvegarde coordonnée des processus, l'état des processus est composé de l'ensemble de ses **chares**. Les sauvegardes d'un processus sont stockées localement et également sur un autre processus qui lui sert de serveur de stockage. Ceci ne permet pas la reprise dans le cas où un processus et son serveur de stockage tombe en panne.

La tableau 2.2 comparent les systèmes détaillés ci-dessus. Les critères de comparaison utilisés sont les suivants.

État sauvegardé : désigne de quelle manière est représenté l'état du processus pour la sauvegarde.

Coordination : indique si les processus se coordonnent au moment de la sauvegarde.

Multithreadé : indique que le mécanisme utilisé permet de sauvegarde des applications multithreadées.

Hétérogénéité : indique si l'état sauvegardé peut-être restauré sur un ensemble varié d'architectures (processeur et système d'exploitation).

Restauration locale ou globale : la reprise est dite globale si le redémarrage nécessite la construction d'un état global cohérent, elle est dite locale si seule la connaissance de l'état du processus (ou éventuellement de ses voisins) est nécessaire.

Remplacement d'une ressource défaillante : précise si on a besoin d'une nouvelle ressource pour remplacer la ressource défaillante ou si on peut utiliser une ressource existante.

Composants fiables quels composants sont supposés fiables pour pouvoir traiter correctement les pannes considérées.

2.7 Conclusion

Ce chapitre a présenté une taxonomie des différentes techniques de tolérances aux pannes qui est résumée à la figure 2.3.

Nous avons vu que les protocoles basés sur la duplication nécessitent un nombre important de ressources au détriment des applications et c'est pourquoi ils ne sont pas adaptés au domaine des applications de calcul numérique. De plus, les protocoles basés sur la duplication ne tolèrent qu'un nombre limité de pannes. Les méthodes de tolérance basées sur une mémoire stable semblent plus adaptées.

Parmi les méthodes basées sur une mémoire stable, on distingue les approches utilisant la journalisation des événements non déterministes et les approches réalisant une sauvegarde de l'état des processus. La journalisation effectue un enregistrement des événements non déterministes sur un support stable. Ces événements sont notamment la réception de messages ; ceci fait que la journalisation est plus adaptée aux applications communiquant peu. Les applications considérées dans le cadre de ce travail communiquent fréquemment. Une approche par journalisation induit un surcoût important et les performances sont nettement dégradées.

La tolérance aux pannes par sauvegarde semble être une approche viable pour les applications de calcul numérique considérées [15, 10]. Cependant, la sauvegarde non coordonnée est affectée par l'effet domino et présente donc le risque de devoir reprendre le calcul au début. La sauvegarde induite par les communications est, comme les approches par journalisation, sensible à la fréquence des communications. La sauvegarde coordonnée se présente comme une bonne solution car elle assure la cohérence

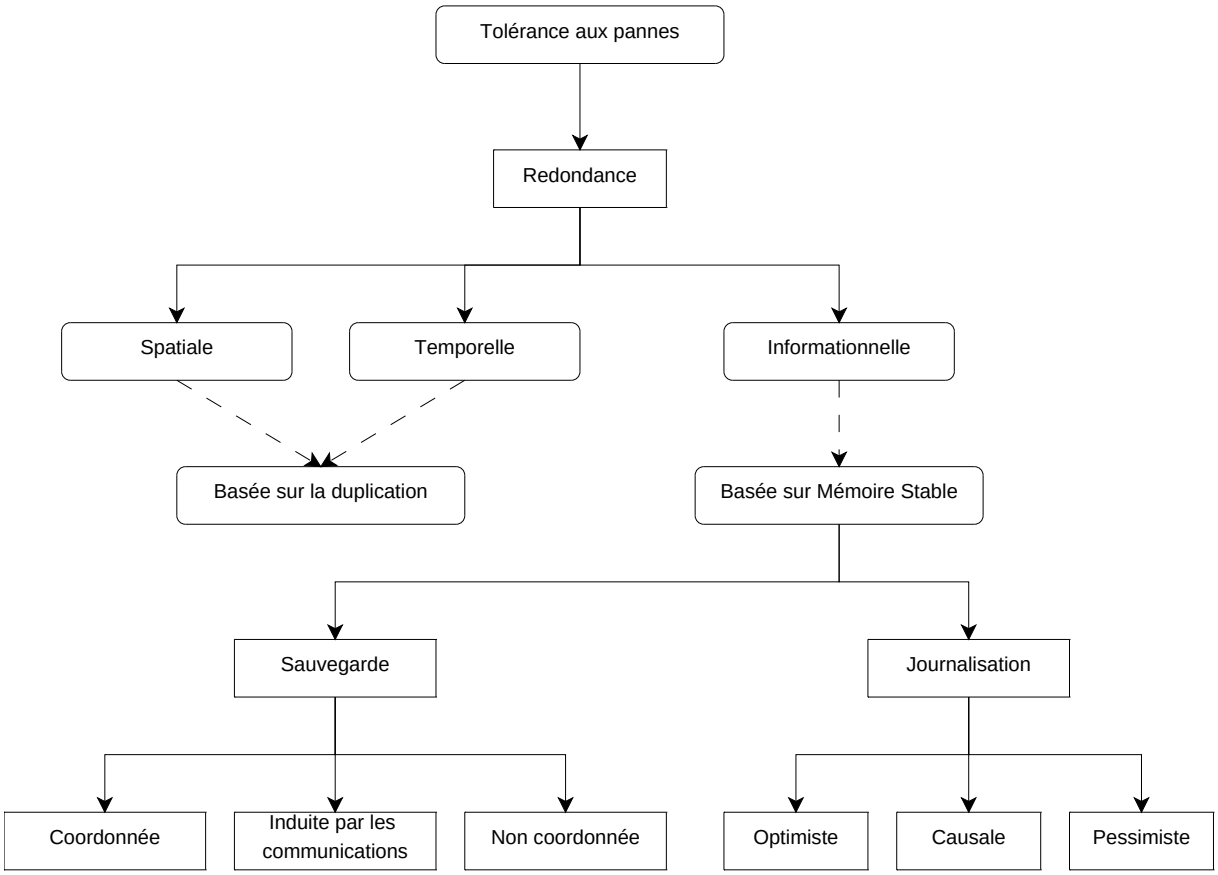


FIG. 2.3 – Tolérance aux pannes par mémoire stable [17]

de l'état global. Les performances actuelles des réseaux et des algorithmes permettent de réaliser cette étape de coordination dans un temps relativement faible sur de nombreuses machines. De plus, ce surcoût induit peut être amorti en choisissant une période de sauvegarde adéquate.

Le chapitre suivant présente un protocole coordonné « standard » qui permettra d'expliquer dans le détail le fonctionnement et les performances d'un tel protocole.

	Journalisation pessimiste	Journalisation optimiste	Journalisation causale	Sauvegarde coordonnée	Sauvegarde non coordonnée	Sauvegarde induite par les comm.
Hypothèse PWD	Oui	Oui	Oui	Non	Non	Non
Effet domino	Non	Non	Non	Non	Possible	Non
Points de reprise par processus	Un	Plusieurs	Un	Un	Plusieurs	Plusieurs
Processus orphelin	Non	Possible	Non	Non	Possible	Possible
Propagation de reprise	Dernier point de reprise	Plusieurs points de reprise	Dernier point de reprise	Dernier point de reprise	Non bornée	Plusieurs points de reprise

TAB. 2.1 – Comparaison des différentes méthodes de tolérance aux pannes par reprise

	CoCheck	MPICH-V1	MPICH-cl	MPICH-causal	Charm++
État sauvegardé	Image mémoire	Image mémoire	Image mémoire	Image mémoire	Chares
Coordination	Oui	Non	Oui	Non	Oui
Multithreadé	Non	Non	Non	Non	Oui
Hétérogénéité	Non	Non	Non	Non	Non
Restauration globale ou locale	Globale	Locale	Globale	Locale	Globale
Remplacement de ressource défaillante	Nouvelle ressource	Nouvelle ressource	Nouvelle ressource	Nouvelle ressource	Ressource existante
Composants fiables	Mémoire stable	Mémoire stable, dispatcher, canal mémoire	Mémoire stable, dispatcher	Mémoire stable, dispatcher	Mémoire stable

TAB. 2.2 – Comparaison des systèmes existants de tolérance aux pannes par reprise

Chapitre 3

Protocole de sauvegarde/reprise coordonnée standard

L'objectif de ce chapitre est de présenter un protocole de tolérance aux pannes par sauvegarde/reprise coordonnée « standard ». La première section présente une modélisation et les hypothèses sur le type d'application considérée, l'environnement d'exécution et le type de pannes à traiter. Puis un protocole standard de sauvegarde/reprise coordonnées est décrit. Il servira de référence pour la comparaison du protocole conçu CCK (Coordinated Checkpointed in KAAPI) qui sera présenté à le chapitre suivant.

3.1 Modèle d'application et hypothèses

Le cadre dans lequel on se place pour l'étude des protocoles de tolérance aux pannes est celui du calcul parallèle. C'est-à-dire que la performance des exécutions des applications est le principal critère à optimiser. Les hypothèses sur les conditions d'exécution sont les suivantes.

L'application :

- Elle s'exécute dans un environnement de type grille, c'est-à-dire que l'ensemble des ressources est dynamique et hétérogène.
- L'objectif principal est de minimiser le temps d'exécution total de l'application.
- On souhaite réaliser un mécanisme de tolérance transparent pour l'utilisateur et le programmeur.

Le modèle d'exécution :

- Les canaux de communication entre les processus sont FIFO (premier envoyé, premier reçu).

Les pannes :

- Le type de panne que l'on souhaite tolérer sont les pannes franches.

3.2 Protocole standard

Cette section décrit un protocole standard de sauvegarde/reprise coordonnée [43] et réalise une analyse de son coût. L'objectif est de fournir une référence qui permette de comparer le protocole que nous proposons dans le chapitre suivant.

3.2.1 Sauvegarde coordonnée

Ce protocole construit un état global cohérent lors de la sauvegarde. Pour cela il utilise, comme présenté dans le chapitre précédent, une étape de synchronisation de tous les processus afin que l'état

global du système se ramène à l'état de chacun des processus. Ceci est réalisé grâce à une opération permettant de garantir que les canaux de communication sont vides et sans état.

Pour réaliser cette sauvegarde, on distingue un processus particulier qui est appelé le processus coordinateur. Ce processus ne participe pas au calcul et son rôle est d'initialiser la sauvegarde et de coordonner les processus durant la sauvegarde.

On note P_0 le processus coordinateur et P_i pour $1 \leq i \leq N$ le processus de calcul i.

L'algorithme du coordinateur est donné à la figure 3.1. Cet algorithme se fait en trois étapes. D'abord (1), le coordinateur initialise l'étape de coordination. Puis (2), il attend que tous les processus aient effectué leur sauvegarde. Enfin, il annonce la fin de la coordination et la reprise des calculs.

Au lancement d'une session de sauvegarde
1. Démarrage de l'étape de sauvegarde coordonnée
1.1. $\forall i \in \{1, .., N\}$*, envoi du message INIT à* P_i

2. Attente de la fin de la sauvegarde des processus
2.1. $\forall i \in \{1, .., N\}$*, attente du message ACK de* P_i

3. Reprise de l'exécution normale
3.1. $\forall i \in \{1, .., N\}$*, envoi du message CONT à* P_i

---

Sur réception d'un message ACK
1. Si le nombre de message ACK reçu est N*, alors réveiller le coordinateur*

FIG. 3.1 – Code du coordinateur du protocole standard

Du coté des processus applicatifs, la fonction de traitement associée au message INIT est développée à la figure 3.2 :

Sur réception d'un message INIT par le processus P_i
1. Arrêt des threads de calcul

2. Vidage des canaux de communication
2.1. $\forall k \in \{1, .., N\}$*, envoi du message FLUSH à* P_k
2.2. $\forall k \in \{1, .., N\}$*, attente du message FLUSH de* P_k

3. Sauvegarde de l'état du processus
3.1. Sauvegarde distante

4. Signalisation de la fin de la sauvegarde
4.1. Envoi du message ACK au coordinateur
4.2 Attente du message CONT du coordinateur

5. Redémarrage des threads de calcul

Sur réception d'un message FLUSH
1. Si le nombre de message FLUSH reçu est N*, alors réveiller le processus*

Sur réception d'un message CONT
1. Réveiller le processus

FIG. 3.2 – Code des processus de calcul du protocole standard

Cet algorithme comporte cinq étapes.

1. L'arrêt des threads : les threads de calcul du processus sont arrêtés. Après cela, aucun message relatif au calcul n'est envoyé.

2. **Le vidage des canaux de communication :** la technique utilisée est la suivante. Les canaux de communication étant supposés FIFO, un message FLUSH est envoyé sur tous les canaux. Une fois tous les messages FLUSH reçus, on est assuré que les canaux de communication sont vides.
3. **La sauvegarde de l'état global du processus :** on sauvegarde l'état du processus sur la mémoire stable.
4. **La signalisation de la fin de la sauvegarde :** pour éviter que la réception d'un message ne perturbe la sauvegarde du processus, on a besoin d'attendre la fin des sauvegardes avant de relancer l'exécution.
5. **Redémarrage des threads de calcul :** le calcul peut reprendre car tous les processus ont terminé leur sauvegarde.

3.2.2 État global cohérent

Propriété 1. *L'ensemble des sauvegardes constituent un état global cohérent de l'application.*

Les canaux de communication sont vides grâce à l'étape de coordination, l'état global se réduit alors à l'état des processus. [43] présente une preuve détaillée.

3.2.3 Reprise

L'état global cohérent de l'application a été construit à la sauvegarde, le principe de la reprise est alors d'effectuer une reprise globale : on repart de l'état global cohérent correspondant à la dernière sauvegarde. Dans ce cas, une condition nécessaire à la reprise est de disposer d'autant de ressources qu'au moment de la dernière sauvegarde. Ainsi pour chaque machine tombée en panne, il est nécessaire d'en allouer une autre. Le redémarrage se fait en trois étapes.

1. On arrête tous les processus.
2. Chaque processus charge un état précédent sauvegardé.
3. Chaque processus reprend le calcul.

3.2.4 Analyse de coût

Nous analysons le coût de ce protocole en terme de coût des sauvegardes puis du coût de la reprise.

Sauvegarde

Le coût de cet algorithme va être mesuré en nombre de messages envoyé. On note N le nombre de processus de calcul.

On fait le décompte pour un processus P_i :

Message INIT :	chaque processus en reçoit 1.
Message FLUSH :	chaque processus en envoie N.
Message ACK :	chaque processus en envoie 1.
Message CONT :	chaque processus en reçoit 1.

On a donc un coût total de $N(N+3) = O(N^2)$. À cela, il faut ajouter le coût d'envoi de N gros messages qui contiennent les sauvegardes des processus.

Concernant le coût en temps de cet algorithme, il peut être négligé en choisissant une période de sauvegarde adéquate. Notons

– $\mathcal{T}_N$ le temps d'exécution du programme parallèle sur N processeurs sans protocole de tolérance aux pannes ;

– $\mathcal{T}_N^{std}$ le temps d'exécution du programme parallèle sur N processeurs avec le protocole standard de tolérance aux pannes ;
– $\mathcal{T}_{sauvegarde}^{std}$ le coût d'une étape de sauvegarde du protocole standard et k le nombre d'étapes de sauvegarde.

On a donc $\mathcal{T}_N^{std} = \mathcal{T}_N + k\mathcal{T}_{sauvegarde}^{std}$ et on peut définir $\tau^{std} = \mathcal{T}_N^{std}/k$ comme la période de sauvegarde. Si l'on souhaite que le surcoût engendré par l'utilisation du protocole soit faible, c'est-à-dire $\mathcal{T}_N^{std} \approx \mathcal{T}_N$ ou $k\mathcal{T}_{sauvegarde}^{std} \ll \mathcal{T}_N$. On choisit donc τ^{std} pour que

$$\tau^{std} \approx \frac{\mathcal{T}_N}{k} \gg \mathcal{T}_{sauvegarde}^{std}$$

Reprise

Dans le cas de la reprise, le coût va être mesuré en comptant le temps nécessaire pour retrouver un état équivalent à celui de l'application juste avant la panne. Ce temps se décompose en plusieurs parties :

$$\mathcal{T}_{reprise}^{std} = \mathcal{T}_{arrêt}^{std} + \mathcal{T}_{chargement}^{std} + \mathcal{T}_{ré-exécution}^{std}$$

avec
– $\mathcal{T}_{arrêt}^{std}$ le temps nécessaire pour arrêter et synchroniser tous les processus ;
– $\mathcal{T}_{chargement}^{std}$ le temps nécessaire pour que chaque processus charge la dernière sauvegarde ;
– $\mathcal{T}_{ré-exécution}^{std}$ le temps nécessaire pour ré-exécuter les tâches perdues, ce sont les tâches qui ont été exécutées entre la dernière sauvegarde et le moment de la panne.

On peut remarquer que le temps d'arrêt et de chargement est du même ordre que le coût d'une étape de sauvegarde, soit $\mathcal{T}_{arrêt}^{std} + \mathcal{T}_{chargement}^{std} \approx \mathcal{T}_{sauvegarde}^{std}$. De même, pour le temps de ré-exécution des tâches perdues, on a $\mathcal{T}_{ré-exécution}^{std} \leq \tau^{std}$. Finalement, $\mathcal{T}_{reprise}^{std} \approx \mathcal{T}_{sauvegarde}^{std} + \tau^{std}$. Le coût en temps d'une reprise est donc

$$\mathcal{T}_{reprise}^{std} = O(\tau^{std})$$

Notons, et il s'agit d'une différence importante avec le protocole CCK que nous proposons dans le chapitre suivant, que ce temps de reprise s'applique à chaque processus de l'application, y compris les processus redémarrés après les défaillances. Le temps séquentiel équivalent $\mathcal{W}_{reprise}^{std}$ (aussi appelé *travail*) suite à la reprise d'un ou plusieurs processus parmi N est :

$$O(N\tau^{std})$$

3.3 Conclusion

Le protocole présenté dans ce chapitre est une version simple du protocole de sauvegarde/reprise coordonnée tel qu'on peut le retrouver dans [43]. De nombreux optimisations sont possibles, tant au niveau de l'étape de sauvegarde qu'au niveau de la phase de reprise.

Dans [24], une optimisation de l'étape de coordination est proposée en ne vidant les canaux de communication qu'avec les processus ayant communiqué depuis la dernière sauvegarde. Dans [15, 10], les auteurs proposent que chaque processus garde une copie en mémoire de la sauvegarde afin d'accélérer le chargement de la dernière reprise.

Dans le chapitre suivant nous montrerons comment utiliser les propriétés de l'intergiciel cible de notre travail. Nous exploiterons sa représentation abstraite afin de réduire la quantité de calcul perdu et le nombre processus qui redémarrent.

Chapitre 4

Protocole de sauvegarde/reprise coordonnée dans un modèle graphe de flot de données

Le protocole présenté dans ce chapitre est appelé CCK (Coordinated Checkpointing in KAAPI). Il a pour objectif d'améliorer les performances du protocole standard en répondant aux questions suivantes :

1. Comment réduire le coût de la synchronisation lors de la sauvegarde ?
2. Comment accélérer le redémarrage des processeurs après une défaillance de l'un d'entre eux ?
3. Comment réduire la quantité de temps d'exécution perdu en cas de panne ?

La solution à la question (1) est inspirée par la sauvegarde coordonnée partielle présentée dans [24] dont l'idée originale est de sauvegarder un ensemble minimal de processus déterminé par les communications effectuées depuis la dernière sauvegarde. Pour le protocole CCK, tous les processus sont sauvegardés, mais la synchronisation se limite au strict minimum : pour un processus donné, seuls les canaux de communication des processus avec lesquels il peut communiquer (les processus dont il attend un message) sont vidés. Cette information peut être extraite du graphe de flot de données qui représente le futur de l'exécution de l'application.

La réponse à la question (2) est proposée dans [15, 10] : chaque processus lors de la sauvegarde conserve une copie locale en plus de la copie envoyée soit à une zone de stockage stable [10] soit à un nombre fixé de processus voisins [15]. De cette manière les processus non défaillants redémarrent en utilisant leur copie locale de la dernière sauvegarde sans avoir besoin de faire un accès distant à la mémoire stable.

La contribution du protocole CCK porte essentiellement sur la solution proposée à la question (3). On désire ne ré-exécuter que les calculs nécessaires pour continuer l'exécution de l'application. En particulier, on souhaite redémarrer depuis leur dernière sauvegarde que les processus défaillants. Pour cela on utilise la représentation abstraite de l'application pour déterminer les calculs strictement nécessaires aux processus défaillants pour reprendre l'exécution. Cet ensemble correspond aux *tâches de calcul* que les processus non défaillants doivent ré-exécuter pour ré-émettre les communications perdues aux processus défaillants.

Ces améliorations (1) et (3) par rapport au protocole standard reposent essentiellement sur la représentation abstraite de l'exécution des applications KAAPI. Cette représentation abstraite est le *graphe de flot de données* et elle est particulièrement bien adaptée à la description d'application parallèle [40]. Les solutions présentées peuvent être adaptées à tout moteur d'exécution utilisant un graphe de flot de données pour représenter l'état d'exécution de l'application.

Un avantage supplémentaire à l'utilisation d'une représentation abstraite est quelle permet d'avoir un format de sauvegarde totalement indépendant de l'architecture. La sauvegarde correspond alors au graphe de flot de données représentant le futur de l'exécution de l'application et à ses entrées, et ceci est

indépendant de l'architecture utilisée. La représentation abstraite de l'application permet de l'utiliser (et en particulier de faire des reprises) dans un environnement hétérogène.

Cette section décrit tout d'abord le modèle de graphe de flot de données et le moteur d'exécution de KAAPI. Ensuite l'étape de sauvegarde coordonnée du protocole CCK, comportant l'amélioration (1), est présenté en 4.2 et l'étape de reprise comportant les améliorations (2) et (3) est détaillée à la section suivante. Ensuite quelques améliorations possibles sont proposées.

4.1 KAAPI : Modèle d'exécution et représentation abstraite

Cette section a pour but de présenter KAAPI. KAAPI [1] est un intergiciel permettant d'exécuter une application parallèle et/ou distribuée. Le modèle d'exécution et la représentation abstraite de KAAPI ont servi de base à la conception du protocole. On présente tout d'abord le modèle de programmation de KAAPI et sa représentation abstraite sous forme de graphe de flot de données, qui est utilisée pour représenter l'état d'exécution d'un programme. Enfin, le moteur d'exécution et l'ordonnancement « statique » sont décrits.

4.1.1 Modèle de programmation

Le modèle de programmation proposé par KAAPI [17] est un modèle de programmation parallèle de haut niveau. Dans cette approche le programme explicite un grand degré de parallélisme potentiel indépendamment de l'architecture. Pour cela, KAAPI utilise une API : Athapascan [16, 38]. Cette API permet de définir le parallélisme selon deux concepts simples : les **tâches** et les **données partagées**. Les tâches constituent un ensemble indivisible d'instructions. Ces tâches peuvent s'exécuter en parallèle. Les données partagées sont des données en mémoire accessibles par les tâches.

Les tâches déclarent des méthodes d'accès (lecture, écriture) sur les données partagées. Ceci permet de définir des dépendances entre les tâches et les données. Ces dépendances décrivent le graphe de flot de données associé à l'exécution de l'application.

4.1.2 Représentation abstraite : graphe de flot de données

Dans KAAPI, l'exécution d'une application est représentée par un graphe de flot de données. Ce graphe est défini de la manière suivante :

Définition 5. *Le graphe de flot de données associé à l'exécution de l'application est le graphe orienté acyclique $G^p = (\mathcal{S}, \mathcal{A})$.*

- *Les tâches $\mathcal{S}_T$ et les données $\mathcal{S}_D$ forment l'ensemble des nœuds du graphe $\mathcal{S} = \mathcal{S}_T \cup \mathcal{S}_D$.*
- *Les accès des tâches aux données forment l'ensemble $\mathcal{A}$ des arêtes du graphe.*

Ce graphe est un graphe biparti : un nœud tâche est connecté à un ou plusieurs nœuds données ; un nœud donnée est connecté à un ou plusieurs nœuds tâches.

Une arête dans le graphe signifie une synchronisation de type lecture ou écriture entre une donnée et une tâche. Soient $t \in \mathcal{S}_T$ et $d \in \mathcal{S}_D$, alors :

- *l'arête $(t, d) \in \mathcal{A}$ signifie que la tâche t accède en écriture à la donnée d ; la tâche t précède toute tâche qui accède à d en lecture.*
- *l'arête $(d, t) \in \mathcal{A}$ signifie que la tâche t accède en lecture à la donnée d ; la tâche t est précédée par toute tâche qui accède à d en écriture.*

Ce graphe est **dynamique** : les nœuds tâches et données sont ajoutés et supprimés du graphe au cous de l'exécution.

D'après [18, 19, 17], la représentation abstraite par graphe de flot de données définie ci-dessus peut-être utilisée pour définir un état global d'une exécution distribuée.

4.1.3 Moteur d'exécution

Lors de l'exécution d'une application, le moteur d'exécution KAAPI réalise dynamiquement la détection du parallélisme de l'application par une analyse des dépendances d'accès aux données partagées. Cette analyse de dépendances est effectué sur la représentation abstraite de l'application sous forme d'un graphe de flot de données. Elle permet à KAAPI lors de l'exécution d'exploiter le parallélisme effectif de l'architecture utilisée pour l'exécution.

La figure 4.1 montre le diagramme d'état d'une tâche avec un ordonnancement par vol de travail. Après sa création, la tâche est dans l'état INIT. Ensuite, le début de l'exécution l'a fait passer dans l'état EXÉCUTION et à la fin de l'exécution elle passe dans l'état TERMINÉE. Les tâches sont détruites lorsqu'elles passent dans l'état TERMINÉE au fur et à messure de leur exécution. À un instant fixé, le graphe de flot de données courant représente donc le futur de l'exécution de l'application.

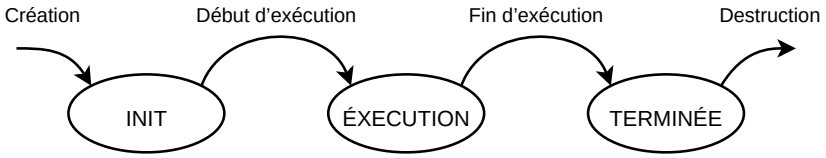


FIG. 4.1 – Diagramme d'état d'une tâche dans KAAPI

La tâche choisie pour l'exécution par le moteur d'exécution KAAPI est la prochaine tâche dans l'état INIT qui respectent les contraintes de synchronisation imposées par le graphe de flot de données.

Un algorithme d'ordonnancement permet de répartir initialement l'ensemble du graphe sur différents processus.

Ordonnancement par vol de travail

Avec un ordonnancement par vol de travail, chaque processeur possède une liste de tâches à exécuter (le graphe de flot de données). Un processeur inactif, c'est-à-dire qu'il n'a plus de tâches prêtes, essaye alors de voler une tâche aux autres processeurs. Le vol peut être local entre deux threads d'un même processus ou distant entre deux processus distincts en passant par le réseaux. L'exécution et le vol de tâches respectent évidemment les contraintes de synchronisation imposées par le graphe de flot de données. Le vol de travail fournit un ordonnancement efficace [16] et le nombre de tâches volées est faible [7, 16]. Ce type d'ordonnancement est particulièrement bien adapté au applications récursives.

4.1.4 Ordonnancement « statique »

L'ordonnancement statique permet de partitionner le graphe de flot de données de l'application pour donner à chaque processeur une partie du graphe à exécuter. KAAPI peut utiliser les librairies SCOTCH [34] ou METIS [23] pour effectuer le partitionnement. L'étape de partitionnement génèrent des tâches supplémentaires dans le graphe de flot de données. Ces tâches représentent les communications qui permettent de transmettre les données nécessaires d'un processus à l'autre.

Une communication est définie par un couple émission-réception et à chaque communication est associé un tag qui l'identifie de manière unique. On peut étendre le graphe de flot de données $G^p = (\mathcal{S}, \mathcal{A})$ donné à la définition 5 en nommant ces tâches de communication.

On définit l'ensemble $\mathcal{C} \subset \mathcal{S}_T$ des tâches de communications. Soit $t \in \mathcal{C}$, t est soit une tâche d'émission, soit une tâche de réception. Un tag sur les tâches de communication permet d'identifier la réception associée à l'émission et inversement.

La figure 4.2 montre le diagramme d'une tâche de réception, dans ce cas, l'exécution correspond à écrire la donnée au bon emplacement mémoire. Le diagramme d'état d'une tâche d'émission est identique à celui d'une tâche classique, l'exécution correspond à l'envoie de la donnée et ne se termine que quand le message a effectivement été émis.

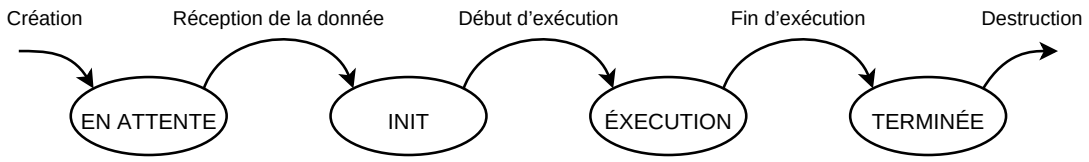


FIG. 4.2 – Diagramme d'état d'une tâche de réception dans KAAPI

La figure 4.3 illustre cette étape d'ordonnancement statique. À gauche un extrait d'un graphe initial est donné. À droite ce même graphe après l'étape d'ordonnancement statique et sa distribution sur trois processus.

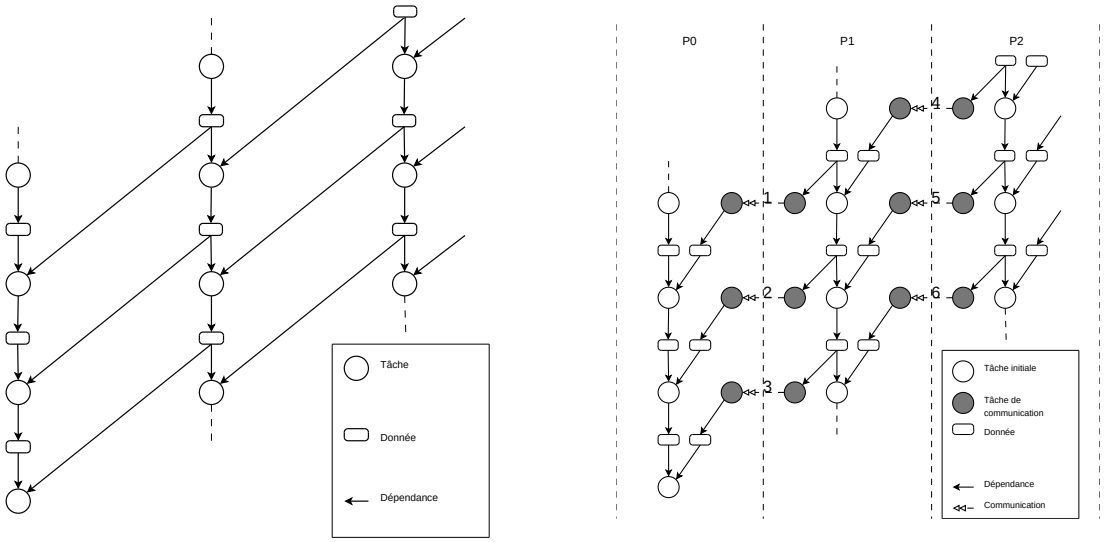


FIG. 4.3 – Exemple de graphe de flot de données. À gauche un extrait d'un graphe initial. À droite le résultat de l'ordonnancement statique et la distribution des sous-graphes sur trois processus.

4.2 Sauvegarde coordonnée

L'état de l'application est représenté par l'état de tous ses processus et par l'état des canaux de communication. Cependant, la connaissance de l'état des canaux de communication n'est pas accessible. Le principe du protocole est de coordonner les processus et de vider les canaux de communication pour effectuer la sauvegarde. L'état de l'application est alors seulement constitué de l'état local de chacun des processus.

4.2.1 Définition d'un point de sauvegarde

L'état d'un processus peut-être sauvegardé grâce à sa représentation abstraite sous forme d'un graphe de flot de données local G_i (qui comprend les tâches et les entrées associées). L'état global de l'application correspond donc à l'ensemble des graphes locaux G_i. Un point de sauvegarde de l'application est donc constitué d'une copie de l'ensemble des graphes de flot de données locaux G_i des processus participant 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.

4.2.2 Mémoire stable

Les points de sauvegarde doivent être stockés sur une mémoire stable. Dans notre cas, la mémoire stable est implémentée par un serveur de sauvegarde distant supposé stable et persistant (pour redémarrer un processeur défaillant) et d'un cache sur le processus qui conserve la dernière sauvegarde et qui permet d'accélérer la reprise sur un processus non défaillant.

4.2.3 Protocole de sauvegarde

Pour garantir la cohérence d'un point de sauvegarde, il est nécessaire de vider les canaux de communication au moment de la coordination. Comme pour le protocole standard, le protocole CCK utilise un processus coordinateur P_0 qui est extérieur au calcul. La différence majeure entre le protocole standard et CCK est que CCK ne va envoyer un message pour vider les canaux de communication qu'entre les processus qui peuvent communiquer. Cette information indiquant si un processus va communiquer avec un autre est tirée du graphe de flot de données.

La figure 4.4 présente l'algorithme du coordinateur. L'algorithme pour le coordinateur présente deux étapes. Tout d'abord le coordinateur initialise une session du protocole. Puis, il attend que tous les processus P_i aient effectué leur sauvegarde. Après ça, le coordinateur peut marquer le point de sauvegarde courant comme étant correct.

Au lancement d'une session de sauvegarde
1. Démarrage de l'étape de sauvegarde coordonnée
1.1. $\forall i \in \{1, .., N\}$*, envoi du message INIT à* P_i

2. Attente de la fin de la sauvegarde des processus
2.1. $\forall i \in \{1, .., N\}$*, attente du message ACK de* P_i

3. Validation du point de sauvegarde
3.1 Dernière_sauvegarde_valide := Sauvegarde_courante

Sur réception d'un message ACK
1. Si le nombre de message ACK reçu est N*, alors réveiller le coordinateur*

FIG. 4.4 – Code du coordinateur du protocole CCK

Pour les processus de calcul, l'algorithme de sauvegarde est activé sur la réception d'un message INIT. Cet algorithme est présenté à la figure 4.5.

Cet algorithme comporte cinq étapes.

1. L'arrêt des threads : il s'agit d'arrêter tous les flots d'exécution internes participants au calcul. L'exécution d'une tâches se faisant de manière non préemptive, le thread est arrêté juste avant l'exécution de la tâche suivante. Après cette étape, on est assuré que le processus n'enverra plus aucun message lié à l'exécution du calcul.

2. Le vidage des canaux de communication : les canaux de communication sont supposés FIFO, l'envoi et la réception d'un message garantissent que tous les messages précédents ont été reçus. Cette étape est un des points clef du protocole CCK qui permet de réduire son coût en nombre de messages.

Plutôt que d'envoyer un message à tous les processus pour vider les canaux de communication, on analyse le graphe de flot de données G_i du processus i qui contient les futures tâches d'émission et de réception. Un message est potentiellement en transit sur le canal de communication avec P_j s'il y a une tâche de réception d'une donnée provenant de P_j dans le graphe. Il faut remarquer que les tâches d'émission ne donnent pas cette information car une fois la tâche d'émission exécutée, elle est détruite et n'apparaît plus dans le graphe alors que le message associé est peut-être toujours en transit sur le canal de communication.

Sur réception d'un message INIT par le processus P_i
1. Arrêt des threads de calcul
1.1 Arrêt des threads de calcul
1.2 Renvoyer un message PONG à P_j pour chaque message PING reçu de P_j

2. Vidage des canaux de communication
2.1. Calcul de $\mathcal{V} = \{$processus dont on attend la réception d'une donnée$\}$
2.2. $\forall k \in \mathcal{V}$, envoi du message PING à P_k
2.3. $\forall k \in \mathcal{V}$, attente du message PONG de P_k

3. Sauvegarde du graphe de flot de données
3.1. Sauvegarde locale et sauvegarde distante

4. Signalisation de la fin de la sauvegarde
4.1. $\forall k \in \mathcal{V}$, envoi du message CONT à P_k
4.2. Attente des messages CONT des processus P_k dont on a reçu un message PING

5. Redémarrage des threads de calcul
5.1 Redémarrage et envoi d'un message ACK au processus coordinateur.

Sur réception d'un message PING
Si les threads sont arrêtés, renvoyer PONG à l'émetteur
Sinon mettre le message en attente (pour l'étape 1.2)

Sur réception d'un message PONG
1. Si le nombre de message PONG reçu est $Card(\mathcal{V})$, alors réveiller le processus

---

Sur réception d'un message CONT
1. Si on a reçu autant de CONT que de PING, alors réveiller le processus

FIG. 4.5 – Code des processus de calcul du protocole CCK

Pour cela, le vidage des canaux de communication est fait par un aller-retour de messages (PING-PONG). Si le processus P_i a une tâche de réception d'une donnée en provenance de P_j dans son graphe de flot de données, alors P_i envoie le message PING à P_j. Sur réception d'un message PING et si le processus P_j a arrêté ses flots d'exécution alors il envoie 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 envoyé à P_i qu'après l'arrêt des threads de calcul. Ceci permet de garantir que plus aucun message lié au calcul ne sera envoyé après le message PONG par l'exécution de tâches de communication.

3. La sauvegarde du graphe de flot de données : les flots d'exécution sont arrêtés et il n'y a plus aucun message en transit, on peut sauvegarder le graphe de flot de données (les tâches et les données en entrées). Cette sauvegarde se fait sur un serveur de sauvegarde et aussi localement (pour permettre éventuellement un redémarrage plus rapide). Un système d'acquittement permet ici de s'assurer que la sauvegarde a bien été reçue et enregistrée.

4. La signalisation de la fin de la sauvegarde : cette étape permet de garantir que la sauvegarde a bien été terminée avant de reprendre les calculs. En effet, il ne faut pas que la réception d'un message lié au calcul perturbe la sauvegarde de l'état du processus. La solution choisie est l'envoi d'un message CONT aux processus dont on peut recevoir des messages liés au calcul. Un processus sait qu'il peut redémarrer lorsqu'il a reçu tous les messages CONT (il doit en recevoir autant que de messages PING). Une autre solution aurait été de différer la délivrance d'un message lié au calcul tant que la sauvegarde n'est pas terminée.

5. Redémarrage des threads de calcul : on reprend le calcul où il a été arrêté. Grâce aux messages CONT, on sait que les processus sont prêts à recevoir les messages qu'on va leur envoyer. On

envoie également un message ACK au processus coordinateur pour l'informer de la fin de la sauvegarde et de la session du protocole.

4.2.4 Preuve de la cohérence globale

Nous allons montrer que l'ensemble des sauvegardes obtenues après une étape de sauvegarde forme un état global cohérent.

L'état global de l'application est défini par :
- l'état de tous les processus ;
- l'état de tous les canaux de communication entre les processus (il faut remarquer que seuls les messages qui sont liés aux calculs effectués par l'application font partie de l'état des canaux de communication).

La sauvegarde réalisée lors d'une étape de sauvegarde ne comporte que les état des processus. Pour montrer que notre sauvegarde est cohérente, nous allons d'abord de montrer que les canaux de communication sont restés vides (de tout message lié au calcul) pendant que les processus ont sauvegardé leur état.

Propriété 2. *Soit le processus P_i. À la fin de l'étape 2 (figure 4.5), les propositions suivantes sont vraies.*

1. *P_i est arrêté.*
2. *Les processus P_j pouvant envoyer un message lié au calcul à P_i sont arrêtés.*
3. *Il n'y a aucun message lié au calcul dans les canaux à destination de P_i.*

Démonstration.

1. P_i est arrêté car il a été arrêté à l'étape 1.
2. Pour chaque message lié au calcul que le processus P_i peut recevoir, P_i possède une tâche de réception correspondante. Il peut calculer l'ensemble $\mathcal{V}_i$ des processus qui peuvent lui envoyer un message lié au calcul. P_i envoie $PING$ à tous les P_j de $\mathcal{V}_i$. P_j ne renvoie un message PONG que quand il est arrêté. Donc quand P_i a reçu tous les messages PONG, tous les processus pouvant envoyer à P_i un message lié au calcul sont arrêtés. On remarque que l'ensemble $\mathcal{V}_i$ ne peut pas changer car le processus P_i est arrêté.
3. Les canaux de communication de communication sont FIFO (First In, First Out), donc quand P_i reçoit un message PONG le canal est vide (car le processus correspondant est arrêté donc il n'envoie plus aucun messsage lié au calcul). Les autres canaux de communication à destination de P_i le sont également car P_i ne peut pas recevoir de messages liés au calcul d'un processus qui n'est pas dans $\mathcal{V}$.

□

Proposition 2. *Soit le processus P_i. Pendant l'étape 3 (figure 4.5),*

1. *Le processus P_i ne reçoit aucun message lié au calcul.*
2. *Les canaux de communication à destination de P_i restent vides.*

Démonstration. D'après la propriété 2, au début de l'étape 3, les canaux de communication sont vides. De plus, les processus P_j pouvant envoyer à P_i un message lié au calcul sont arrêtés. Les canaux de communication à destination resteront vides jusqu'au redémarrage des processus P_j.

Or, pour redémarrer, les processus P_j attendent la réception du message CONT envoyé par P_i. Ce message n'est envoyé qu'à l'étape 4. Donc le processus ne recevra aucun message lié au calcul avant l'étape 4 et donc les canaux de communications resteront vides durant toute l'étape 3.

□

On peut alors montrer que l'état global sauvegardé est cohérent.

Proposition 3. *L'état global sauvegardé est un état global cohérent.*

Démonstration. D'après la proposition 2, les canaux de communication sont restés vides de tout message lié au calcul pendant que les processus ont sauvegardé leur état. Par conséquent, pour toute tâche d'émission marquée comme exécutée dans le graphe local d'un processus, la tâche de réception correspondante a bien reçu la donnée.

On en déduit qu'il n'y a pas de processus orphelin. L'état sauvegardé est donc un état global cohérent. □

4.2.5 Analyse de complexité

Le coût de la sauvegarde coordonnée avec CCK va être mesuré en comptant le nombre de messages envoyés durant l'étape de sauvegarde.

On utilise les notations suivantes :

- N est le nombre totale de processus participant au calcul et donc à la coordination.
- V_i est le nombre de voisins du processus P_i, c'est-à-dire le nombre de processus dont on attend la réception d'une donnée.
- V est le nombre moyen de voisins par processus : $V = \frac{1}{N}\sum_{i=1}^{N} V_i$

On fait le décompte pour un processus P_i :

Message INIT :	chaque processus en reçoit 1.
Message PING :	chaque processus en envoie V_i.
Message PONG :	chaque processus en reçoit V_i.
Message CONT :	chaque processus en reçoit V_i.
Message ACK :	chaque processus en envoie 1.

D'où le nombre total de messages :

$$\sum_{i=1}^{N}(3V_i + 2) = 2N + 3\sum_{i=1}^{N} V_i = 2N + 3NV = O(NV)$$

Pour le type d'applications considérées, comme les simulations de phénomènes physiques, le nombre de voisins d'un processus est en général petit par rapport à N et fixé pour une application donnée. Ce nombre dépend du modèle physique de la simulation. Par exemple pour un modèle physique à deux dimensions où un processus effectue le calcul correspondant à une cellule de forme carrée, le nombre de voisins est 4 si les interactions se font uniquement par les cotés du carré ; il est de 8 si on prend aussi en compte les cellules voisines en diagonale.

À cela il faut rajouter N gros messages correspondant aux sauvegardes des N processus à destination des serveurs de sauvegarde.

Du point de vue du coût en temps de cet algorithme, comme pour le protocole standard, on peut négliger le surcoût en choisissant une période de sauvegarde adéquate. Nous donnerons des mesures expérimentales de ce temps dans la section 5.3.1. On note

- $\mathcal{T}_N$ le temps d'exécution du programme parallèle sur N processeurs sans protocole de tolérance aux pannes ;
- $\mathcal{T}_N^{CCK}$ le temps d'exécution du programme parallèle sur N processeurs avec le protocole CCK ;
- $\mathcal{T}_{sauvegarde}^{CCK}$ le coût d'une étape de sauvegarde du protocole et k le nombre d'étapes de sauvegarde.

Si on se réfère aux calculs effectués pour le protocole standard, on peut avoir $\mathcal{T}_N^{CCK} \approx \mathcal{T}_N$ si on choisit $\tau^{CCK} = \mathcal{T}_N^{CCK}/k \gg \mathcal{T}_{sauvegarde}^{CCK}$.

4.2.6 Comparaison avec le protocole standard

La sauvegarde avec le protocole CCK se différencie essentiellement au niveau de nombre de messages envoyés pour la synchronisation. En effet, le protocole ne vide les canaux de communication qu'entre les processus pour lesquels cela est strictement nécessaire.

Ainsi on a une coordination en $O(NV)$ pour le protocole CCK alors qu'elle se fait en $O(N^2)$ pour le standard avec N le nombre de processus qui participent au calcul et V le nombre moyen de processus voisins. Ceci est intéressant car V est généralement petit devant N (et dans tous les cas $V \leq N$).

La réduction du nombre de messages permet de réduire la congestion du réseau et ainsi de diminuer la durée d'une étape de synchronisation. Tout ceci devrait permettre un meilleur passage à l'échelle du protocole. Ceci sera confirmé par les expériences réalisées dans la section 5.3.3.

4.3 Reprise

On désire redémarrer l'application après la panne de x processus. Juste après la panne, l'application comporte deux types de processus : des processus arrêtés (ou défaillants) et des processus en cours d'exécution (non défaillants).

Quel que soit le processus, sa dernière sauvegarde est stockée sur un support stable. L'ensemble des sauvegardes pour tous les processus constitue un état global cohérent de l'application. De plus, pour les processus encore en exécution, on peut accéder à l'état du calcul en cours.

4.3.1 Problèmes

La méthode simple du protocole standard consiste à redémarrer tous les processus à partir de leur dernière sauvegarde puisque l'ensemble des sauvegardes constitue un état global cohérent. Mais dans ce cas, les calculs réalisés sur tous les processus depuis la dernière sauvegarde sont perdus. Pour le redémarrage avec le protocole CCK, les processus défaillants redémarrent à partir de leur dernière sauvegarde et les processus non défaillants conservent leurs calculs en cours. Cet état global n'est pas cohérent donc on demande également aux processus non défaillants de ré-exécuter une partie des tâches de leur dernière sauvegarde de manière à obtenir un état global cohérent. Ceci réduit le nombre de tâches à ré-exécuter et permet au processus non défaillants de conserver le bénéfice des calculs déjà effectués.

Ces dépendances entre les processus s'expriment à travers les communications. Si une communication d'un processus non défaillant à un processus défaillant a déjà été exécutée, on va demander au processus non défaillant de refaire les calculs permettant de rejouer cette communication. Une des difficultés de cette approche réside dans le fait que rejouer une telle communication peut entraîner également la nécessité de rejouer d'autres communications, en particulier sur des processus ne communiquant pas directement avec les processus défaillants.

Cette section décrit la méthode utilisée dans CCK pour calculer l'ensemble des communications à rejouer et des calculs à refaire sur chaque processus pour garantir un état global cohérent lorsque seuls les processus défaillants sont redémarrés à partir de leur dernière sauvegarde. Cette section comprend aussi une preuve de correction de cette méthode et une analyse de coût en comparaison avec le protocole standard.

Le figure 4.6 montre un exemple d'état d'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 a bien entendu perdu toutes les tâches qu'il avait exécuté.

4.3.2 Notations

Dans la suite, on note X^p un graphe ou un ensemble associé au processus p et X un graphe ou un ensemble associé à tous les processus. X correspond à la réunion des X^p pour tous les processus p. Ces notations reprennent les définitions vues dans la section 4.1

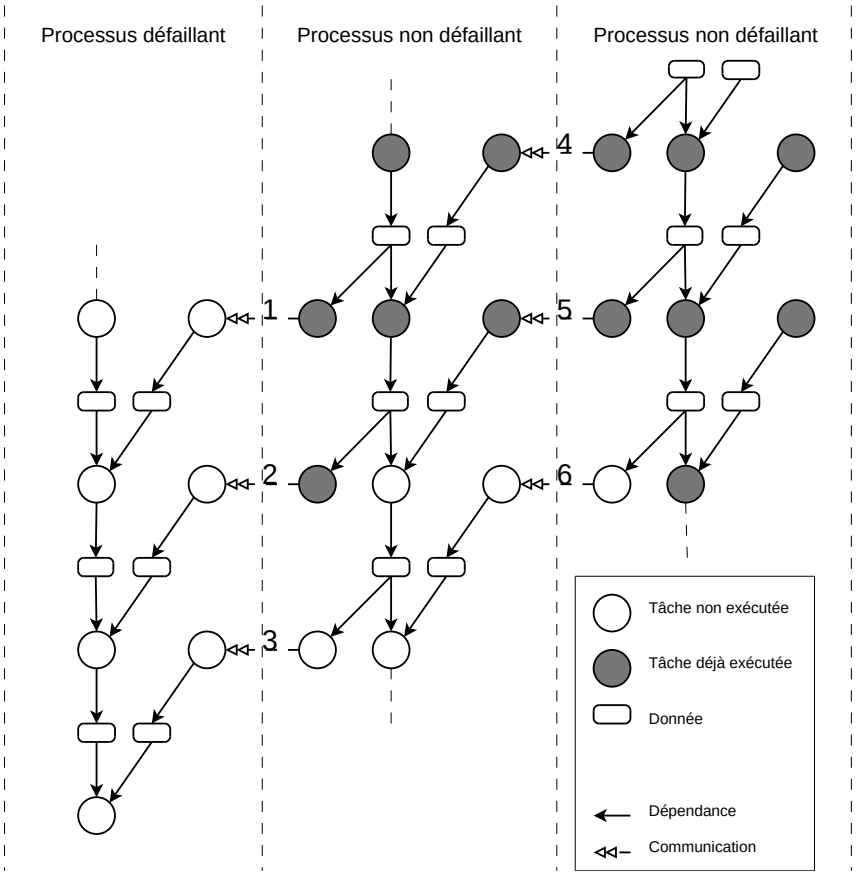


FIG. 4.6 – Exemple avec un processus défaillant et deux processus non défaillants ; les tâches déjà exécutées sont marquées

Soit $G = (\mathcal{S}, \mathcal{A})$ un graphe avec $\mathcal{S}$ l'ensemble des sommets et $\mathcal{A}$ l'ensemble des arêtes. On note $G^* = (\mathcal{S}, \mathcal{A}^*)$ le fermeture transitive de ce graphe.

Graphe de flot de données G^p

La sauvegarde d'un processus p est constitué du graphe de flot de données G^p et de ses données en entrée.

Un graphe de flot de données est un graphe orienté acyclique $G^p = (\mathcal{S}, \mathcal{A})$. C'est aussi un graphe biparti entre les tâches ($\in \mathcal{S}_T$) et les données ($\in \mathcal{S}_D$) (on a $G = \mathcal{S}_T \cup \mathcal{S}_D$). Chaque sommet tâche est connecté à un ou plusieurs sommets données et chaque sommet donnée est connecté à un ou plusieurs sommets tâches comme présenté à la sous section 4.1.2.

Tâches de communication

Parmi les sommets tâches, on trouve les tâches de communication. Une tâche de communication peut être soit une émission, soit une réception. On notera $\mathcal{C}$ le sous-ensemble de $\mathcal{S}_T$ 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 tag unique. On remarque 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^p.

On désigne une communication par son tag t. Soit t une communication, $émission(t) \in \mathcal{S}_T$ est la tâche d'émission associée à la communication t et $réception(t) \in \mathcal{S}_T$ est la tâche de réception associé

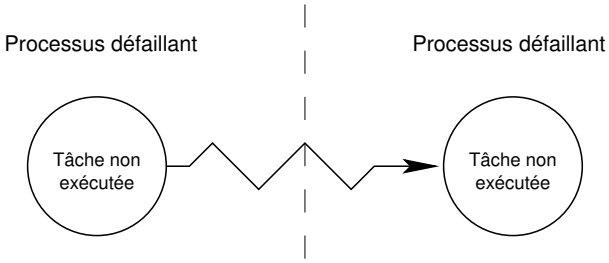
à la communication t. Réciproquement, si c est une tâche de communication (émission ou réception), $tag(c)$ est le tag correspondant.

4.3.3 Ensemble des communications perdues $\mathcal{C}_{perdues}$

On cherche à déterminer les communications nécessaires pour rétablir l'application dans un état cohérent. Plusieurs cas de figure peuvent se présenter selon l'état des processus qui communiquent :

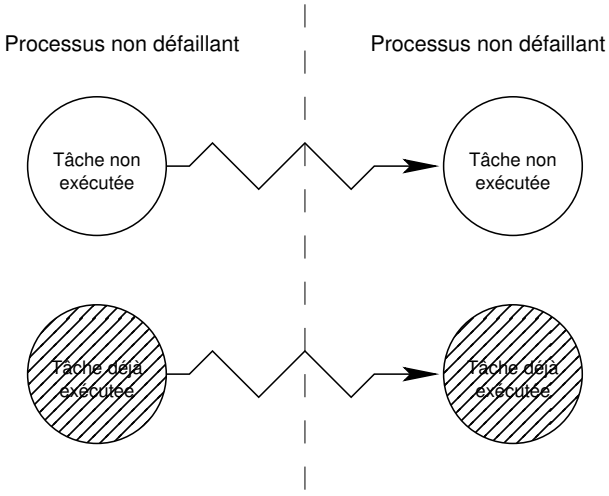
Communication entre deux processus défaillants.

Les deux processus redémarrent à partir de leur dernière sauvegarde, toutes les tâches de communication entre les deux processus seront rejouer lors de la reprise. Il n'y a donc rien à rejouer.



Communication entre deux processus non défaillants.

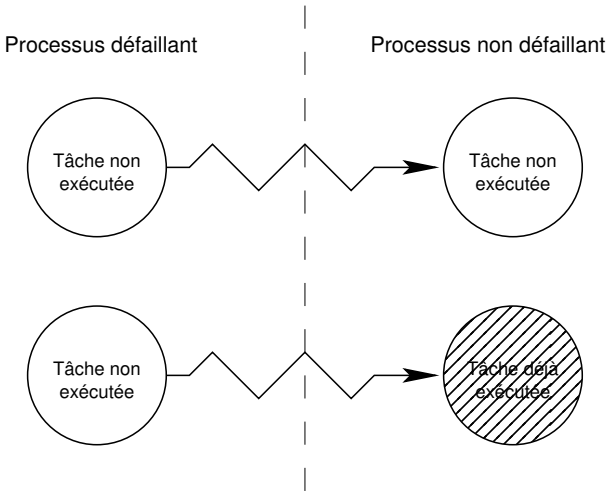
Les deux processus ont exécutés des tâches de communication. Cependant l'état de ces processus résulte d'une exécution normale, les communications entre ces deux processus sont dans un état cohérent. Il n'y a rien à rejouer.



Communication d'un processus défaillant vers un processus non défaillant.

Le processus défaillant redémarre à partir de sa dernière sauvegarde et il va donc ré-émettre toutes les communications de son graphe. Pour le processus non défaillant, on doit distinguer deux cas selon l'état de la réception correspondante :

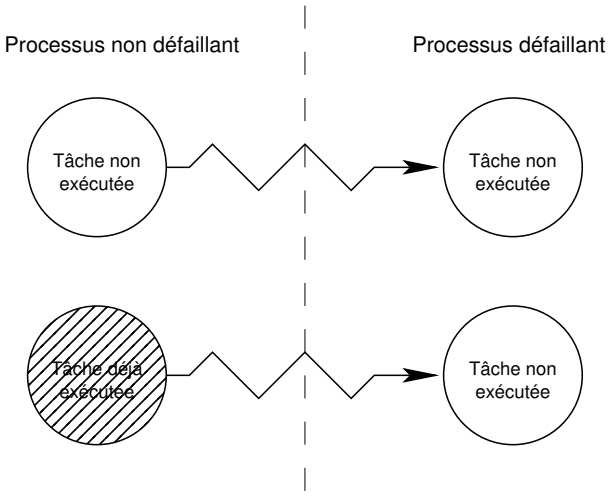
- Si la réception n'a pas été effectuée, ceci est cohérent avec l'état de l'émission. Il n'y a rien à rejouer.
- Si la réception a déjà été effectuée, l'état est incohérent. Cependant cette communication n'est pas utile au processus non défaillant pour continuer son calcul. Dans ce cas, le processus défaillant ne doit pas ré-émettre 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 ré-exécuter 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 6. *On appelle l'ensemble des communications répondant à cette condition l'ensemble des communications perdues et on le note* $\mathcal{C}_{perdues}$.

Cet ensemble sera également appelé ensemble des communications orphelines car une telle communication dénote la présence d'un processus orphelin.

Sur la figure 4.7, on peut voir l'ensemble des communications perdues pour notre exemple ; ce sont les communications 1 et 2.

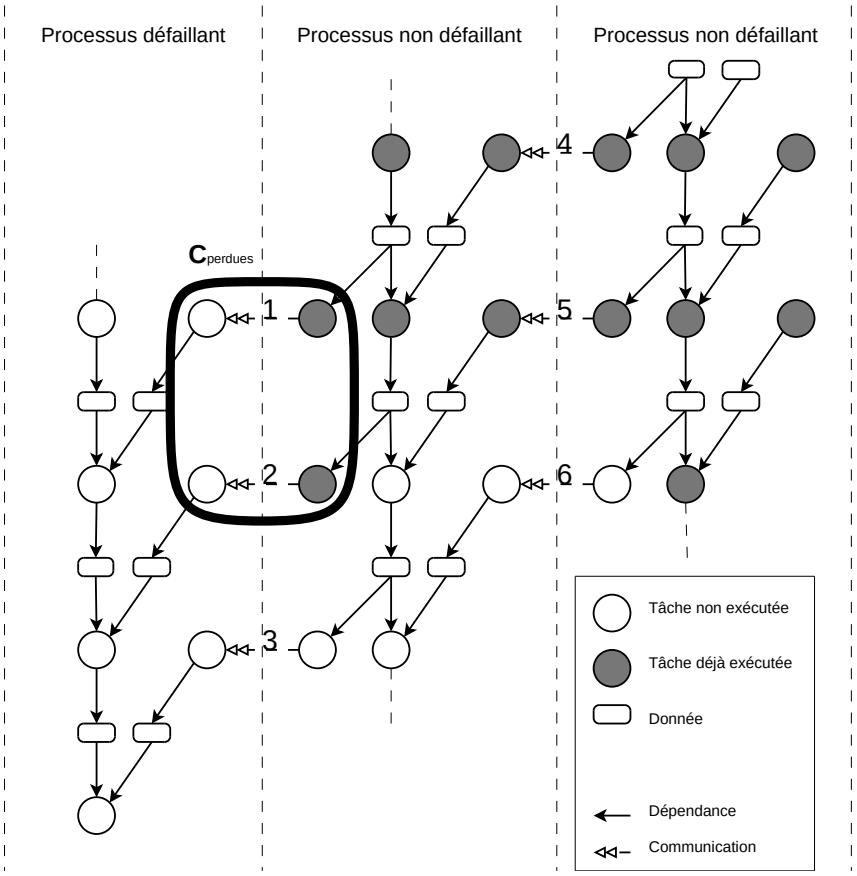


FIG. 4.7 – L'ensemble des communications perdues

4.3.4 Algorithme

On propose ici un algorithme qui implémente le calcul des tâches à réexécuter. Cet algorithme s'exécute localement sur chaque processus, défaillant ou non, lors d'un redémarrage. Il travaille sur le graphe de la dernière sauvegarde sur lequel sont ajoutées les informations concernant l'état d'exécution courant.

1. Arrêt des threads et vidage des canaux de communication
2. Construction du graphe restreint aux communications $\overline{G^p}$
3. Diffusion de $\overline{G^p}$ et de l'état de chaque communication à tous les processus
4. Calcul de l'ensemble $\mathcal{C}_{perdues}$ des communications perdues
5. Calcul de l'ensemble $\mathcal{C}^p_{total}$ des communications à rejouer
6. Calcul de l'ensemble $\mathcal{T}^p_{a\_ré-exécuter}$ des tâches à ré-exécuter
7. Redémarrage des threads

Les parties suivantes définissent les graphes et les ensembles utilisés dans l'algorithme.

Graphe restreint aux communications $\overline{G^p}$

On définit le graphe restreint aux communications $\overline{G^p}$ qui représente les dépendances entre les émissions et les réceptions de données au sein d'un processus p. Ce graphe permet de calculer l'ensemble des données à recevoir pour calculer de manière globale l'ensemble des tâches à rejouer pour ré-émettre les communications perdues vers le processus défaillant.

Définition 7. *On définit le graphe restreint aux communications $\overline{G^p} = (\overline{\mathcal{S}^p}, \overline{\mathcal{A}^p})$ tel que :*
Les sommets $\overline{\mathcal{S}^p}$ sont les tâches de $\mathcal{S}^p$ qui sont des communications ($\overline{\mathcal{S}^p} = \mathcal{C}^p$).
Les arcs $\overline{\mathcal{A}^p}$ sont définies de la manière suivante : $\forall i, j \in \overline{\mathcal{S}^p}$, $(i, j) \in \overline{\mathcal{A}^p}$ si et seulement si

– *i est une réception,*
– *j est une émission,*
– *et s'il existe un chemin de i à j.*

Une définition équivalente de $\overline{G^p}$ est que le graphe restreint aux communications $\overline{G^p}$ est le sous-graphe induit par les sommets des tâches de communication de la fermeture transitive de G^p.

Ainsi, pour $i, j \in \overline{\mathcal{S}^p}$, l'émission de j par p nécessite d'abord la réception de i par p si et seulement si il existe dans $\overline{G^p}$ un arc allant de i à j.

On définit le graphe restreint global $\overline{G} = \bigcup_p \overline{G^p}$. Le graphe $\overline{G}$ représente les dépendances entre les communications entre tous les processus.

Ensemble des tâches à ré-exécuter $\mathcal{T}^p_{a\_ré-exécuter}$

L'ensemble $\mathcal{C}_{perdues}$ 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 ré-exécuter 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 $\overline{G} = \bigcup_p \overline{G^p}$ permet, grâce aux dépendances entre les communications qu'il contient, de déterminer l'ensemble total $\mathcal{C}_{total}$ des communications à rejouer pour ré-émettre toutes les communications perdues.

Formellement, si $\overline{G}^* = (\overline{\mathcal{S}}, \overline{\mathcal{A}}^*)$ est la fermeture transitive de $\overline{G} = \bigcup_p \overline{G^p}$, on a

$$\mathcal{C}_{total} = \mathcal{C}_{perdues} \cup \bigcup_{c \in \mathcal{C}_{perdues}} \{tag(i) \text{ tels que } \exists \in \overline{\mathcal{S}} \text{ et } (i, émission(c)) \in \overline{\mathcal{A}}^*\}$$

On définit alors $\mathcal{C}^p_{total}$ comme l'ensemble des communications à rejouer appartenant au processus p.

Sur la figure 4.8, on peut voir l'ensemble totale des communications à rejouer ; ce sont les communications 1, 2 et 4. La communication 4 doit être rejouée car la communication 2 dépend de 4.

Une fois que chaque processus connaît son ensemble $\mathcal{C}^p_{total}$, il peut déterminer l'ensemble $\mathcal{T}^p_{a\_ré-exécuter}$ des tâches de calcul à ré-exécuter. Ce sont les tâches dont dépendent les communications de $\mathcal{C}^p_{total}$. L'ensemble $\mathcal{T}^p_{a\_ré-exécuter}$ contient l'intégralité des tâches à ré-exécuter pour le processus p. En effet, si une de ces tâches de calcul nécessite une communication, elle a été prise en compte dans l'ensemble $\mathcal{C}^{p'}_{total}$ du processus voisin p' grâce au calcul de la fermeture transitive G^*.

Si $G^* = (\mathcal{S}, \mathcal{A}^*)$ est la fermeture transitive de G, on a

$$\mathcal{T}^p_{a\_ré-exécuter} = émission(\mathcal{C}_{total}) \cup \bigcup_{c \in \mathcal{C}^p_{total}} \{i \in \mathcal{S}^p \text{ tels que } (i, émission(c)) \in \mathcal{A}^*\}$$

Sur la figure 4.9, on peut voir l'ensemble des tâches à ré-exécuter pour reprendre le calcul. Ce sont toutes les tâches nécessaires pour rejouer les communications de $\mathcal{C}_{total}$. $G_{défaillant}$ est le graphe des processus défaillants ; ses tâches doivent également être ré-exécutées.

4.3.5 Preuve de correction de la reprise

Cette partie montre que l'algorithme de reprise proposé est correcte. Pour cela on montre que l'on aboutit à un état global cohérent. On rappelle plusieurs points importants :
– L'état correspondant à l'ensemble des dernières sauvegardes est un état global cohérent (d'après 4.2.4).
– L'application est arrêtée lors des calculs sur les graphes G^p (les graphes sont donc figés).
– Les canaux de communications sont vides lors du calcul de l'ensemble $\mathcal{C}_{perdues}$ des communications perdues.

Propriété 3. *Durant toute la phase de reconstruction de l'état global, les canaux de communication sont vides.*

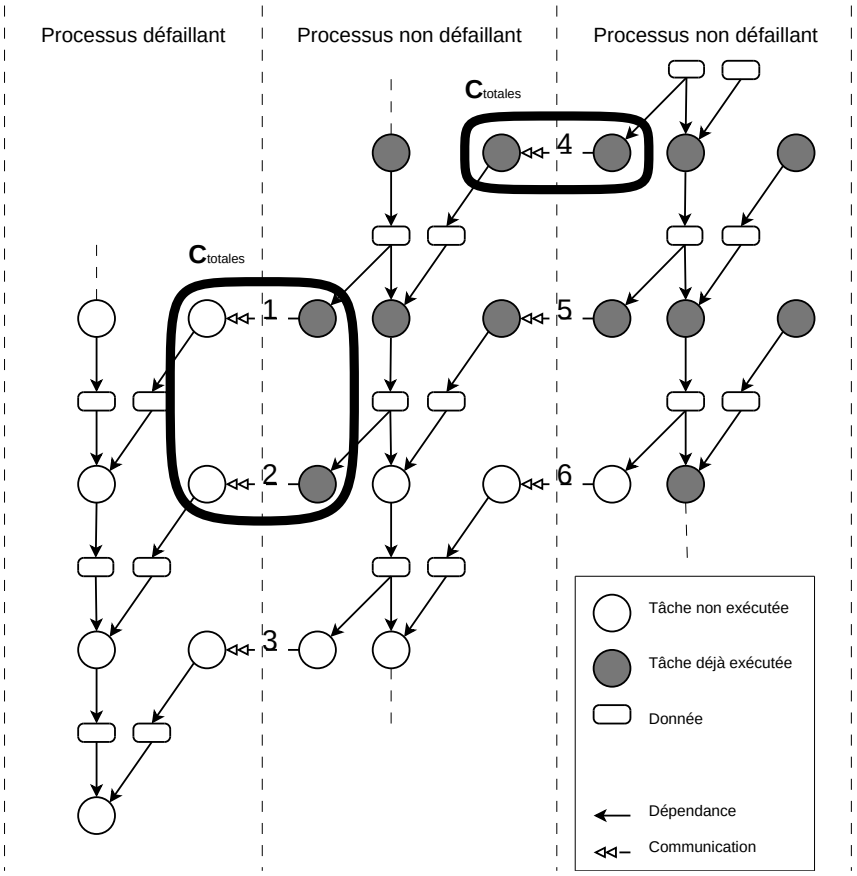


FIG. 4.8 – L'ensemble total des communications à rejouer

Démonstration. Au début de la phase de reprise, on effectue l'arrêt des threads et le vidage des canaux de communication d'une manière similaire à celle présentée pour la sauvegarde. D'après la proposition 3, les canaux restent donc vides jusqu'aux messages indiquant le redémarrage.

□

Définition 8. *On définit l'**état global reconstruit** comme l'état de l'application à la fin de la phase de reprise au moment de redémarrer les threads.*

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 $\mathcal{T}_{a\_ré-exécuter}$ des tâches à ré-exécuter.*

De plus, d'après la propriété 3, les canaux sont vides. Leur état n'intervient donc pas dans l'état global reconstruit.

Notons que $G_{exécution}$ et $G_{défaillant}$ sont des graphes de flot de données.

Proposition 4. *L'état global reconstruit est un état global cohérent.*

Démonstration. Montrons que l'état global reconstruit ne comporte pas de processus orphelin. Dans notre cas, les processus orphelin sont caractérisés par une communication orpheline, c'est-à-dire, une communication qui a une tâche de réception sans tâche d'émission associée.

Nous avons défini $\mathcal{C}_{perdues}$ comme l'ensemble des communications orphelines du graphe $G_{exécution} \cup G_{défaillant}$.

Nous allons montrer que :

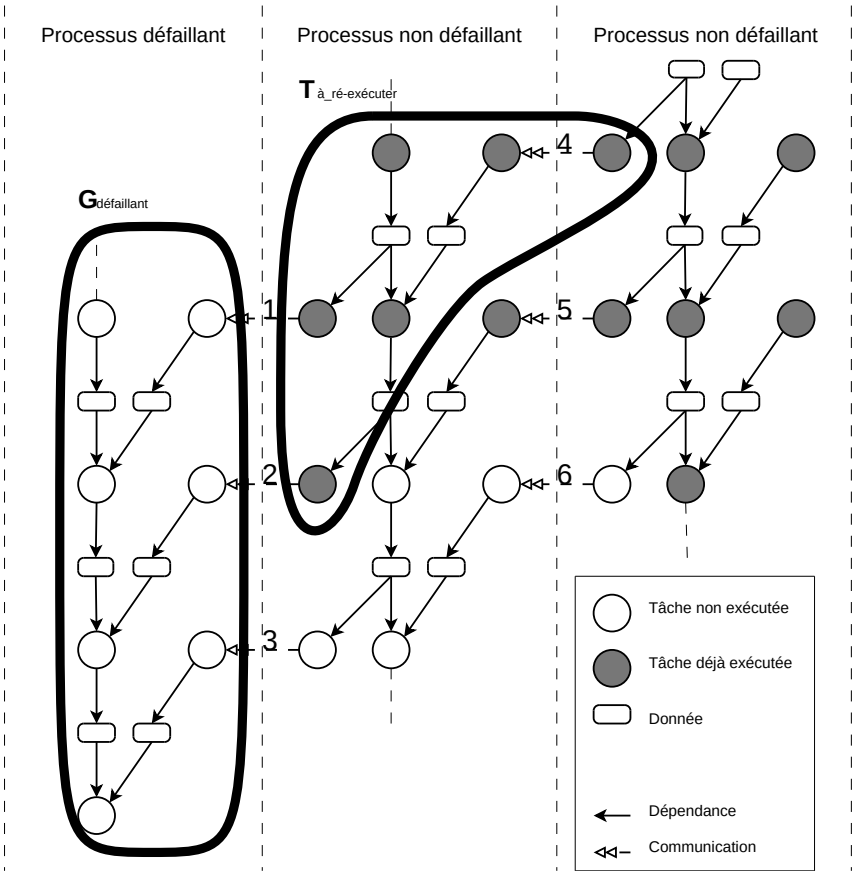


FIG. 4.9 – L'ensemble des tâches à ré-exécuter

1. les tâches d'émissions des communications orphelines $\mathcal{C}_{perdues}$ sont dans $\mathcal{T}_{a\_ré-exécuter}$;
2. l'ensemble $\mathcal{T}_{a\_ré-exécuter}$ ne comporte pas de réception sans émisson.

Tout d'abord, on a bien $émission(\mathcal{C}_{perdues}) \subset \mathcal{T}_{a\_ré-exécuter}$ car d'après les définitions de $\mathcal{C}_{total}$ et $\mathcal{T}_{a\_ré-exécuter}$, on a $émission(\mathcal{C}_{perdues}) \subset \mathcal{C}_{total}$ et $émission(\mathcal{C}_{total}) \subset \mathcal{T}_{a\_ré-exécuter}$.

De plus, l'ensemble des communications de $\mathcal{T}_{a\_ré-exécuter}$ est $\mathcal{C}_{total}$ 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 cohérent d'après la proposition 3 ce qui garantit qu'il n'y a pas de communication orpheline. De plus, d'après la définition de $\mathcal{C}_{total}$, si une communication $c \in \mathcal{C}_{total}$ alors $émission(c) \in \mathcal{T}_{a\_ré-exécuter}$.

Tout ceci nous garantit que l'état global reconstruit est un état global cohérent. □

4.3.6 Analyse du coût

Arrêt des threads et vidage des canaux de communication

Cette étape se fait de la même manière que la coordination de la sauvegarde. Son coût est donc en $O(VN)$ messages (N est le nombre de processus, V le nombre moyen de processus voisins) comme cela a été montré à la section précédente.

Graphe restreint aux communications

Le graphe G^p sur lequel on travaille est un graphe orienté acyclique. Le calcul de la fermeture transitive peut donc se faire en $O(|\mathcal{S}^p| + |\mathcal{A}^p|)$ [12]. De même pour l'extraction du sous-graphe. De plus, ces calculs sont faits en parallèle, chaque processus p calcul son graphe $\overline{G^p}$ à partir de G^p. Le coût du calcul du graphe restreint est donc en $O(max_p(|\mathcal{S}^p| + |\mathcal{A}^p|))$.

Diffusion des graphes restreints

Cette étape effectue la diffusion de tous les graphes $\overline{G^p}$ à tous les processus. Cette communication collective est un schéma du type *Gather-to-All*. Ce type de communication peut être réalisé avec un coût de $O(log_2 N)$ messages [35, 45].

Ensemble des communications perdues

Pour déterminer l'ensemble $\mathcal{C}_{perdues}$ des communications perdues, il suffit de parcourir le graphe global $\overline{G}$ et de regarder l'état des tâches de communications correspondantes (émissions et réceptions).

Le coût de ce calcul est donc en $O(|\overline{\mathcal{S}}|)$.

Ensemble total des communications à rejouer

La détermination de cet ensemble $\mathcal{C}_{total}$ 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 $O(|\overline{\mathcal{S}}| + |\overline{\mathcal{A}}|)$ [12].

Ensemble des tâches à ré-exécuter

L'ensemble $\mathcal{T}^p_{a\_ré-exécuter}$ des tâches à ré-exécuter sur le graphe G^p sont les tâches dont dépendent les communications de ${\mathcal{C}_{total}}^p$. La détermination de cet ensemble peut encore se faire par le calcul de la fermeture transitive du graphe G^p qui est orienté et acyclique qui a déjà été effectué pour le calcul du graphe restreint.

Redémarrage des threads

Cette étape consiste juste à l'envoi d'un message à chaque processus. Ceci est fait en N messages.

Coût total

Le coût en temps de cet algorithme permettant le calcul de l'ensemble des tâches à ré-exécuter est donc en $O(|\overline{\mathcal{S}}| + |\overline{\mathcal{A}}|)$. Le coût en nombre de messages pour cet algorithme est en $O(NV)$.

4.3.7 Travail perdu

On évalue aussi le travail perdu à cause de la panne.

$$\mathcal{T}^{cck}_{reprise} = \mathcal{T}^{cck}_{arrêt} + \mathcal{T}^{cck}_{chargement} + \mathcal{T}^{cck}_{calcul} + \mathcal{T}^{cck}_{ré-exécution}$$

avec
- $\mathcal{T}^{cck}_{arrêt}$ le temps nécessaire pour arrêter et synchroniser tous les processus ;
- $\mathcal{T}^{cck}_{chargement}$ le temps de chargement de la dernière sauvegarde ;
- $\mathcal{T}^{cck}_{calcul}$ le temps nécessaire pour calculer l'ensemble de tâches à ré-exécuter ;
- $\mathcal{T}^{cck}_{ré-exécution}$ le temps nécessaire pour ré-exécuter les tâches perdues, c'est-à-dire les tâches de $G_{défaillant}$ et de $\mathcal{T}_{a\_ré-exécuter}$.

On peut remarquer que le temps d'arrêt est du même ordre que le coût d'une étape de sauvegarde, soit $\mathcal{T}^{cck}_{arrêt} \approx \mathcal{T}^{cck}_{sauvegarde}$. Si les processus ont conservé une copie locale de leur dernière sauvegarde, le temps de chargement $\mathcal{T}^{cck}_{chargement}$ devient négligeable. Pour les processus défaillants, la copie locale n'est plus accessible, on a donc $\mathcal{T}^{cck}_{chargement} \approx \mathcal{T}^{cck}_{sauvegarde}$.

De même, pour le temps de ré-exécution des tâches perdues, on a $\mathcal{T}^{cck}_{ré-exécution} \leq \tau^{cck}$. Cette borne supérieure est similaire au cas de la reprise standard. Des expériences permettront de mettre en évidence ce phénomène dans le chapitre suivant. On suppose également que $\mathcal{T}_{calcul} \ll \tau^{cck}$, car sinon la reprise

CCK devient aussi coûteuse que le temps de calcul à rejouer. Cette hypothèse sera également vérifiée expérimentalement.

Notons que les processus défaillants requièrent un temps $\mathcal{T}^{cck}_{reprise} = O(\tau^{cck})$ sous les hypothèses précédentes. Les processus voisins des processus défaillants doivent ré-exécuter les tâches perdues. Bien que dépendant du flot de données et donc des applications, le protocole CCK n'implémente qu'un redémarrage partiel de l'ensemble des processus : du fait que certains des processus peuvent ne pas avoir à ré-exécuter des tâches, ceux-ci sont libres afin de redistribuer la charge de calcul dû à la ré-exécution des tâches perdues.

On peut également calculer le coût en travail. Soit $N_{défaillants}$ le nombre de processus défaillants.

– Les processus défaillants redémarrent à partir de leur dernière sauvegarde. Leur travail est donc pour chacun $\mathcal{W}^{def}_{reprise} = O(\tau^{cck})$.
– Les processus non défaillants doivent rejouer les tâches de $\mathcal{T}^{p}_{ré-exécution}$. En pratique, pour un processus p, si p est un voisin proche des processus défaillants (au sens de la relation de voisinage définie plus haut, c'est-à-dire par les communications), on aura $\mathcal{W}^{p}_{reprise} \leq \varepsilon^p \tau^{cck}$ (avec $\varepsilon^p \leq 1$ qui représente le pourcentage de tâches à ré-exécuter sur le processus p) ; sinon $\mathcal{W}^{p}_{reprise} \approx 0$.

Le coût en travail est donc

$$\mathcal{W}^{cck}_{reprise} = O((N_{défaillants} + \varepsilon)\tau^{cck})$$

Ce travail nécessaire pour la reprise est illustré à la figure 4.10

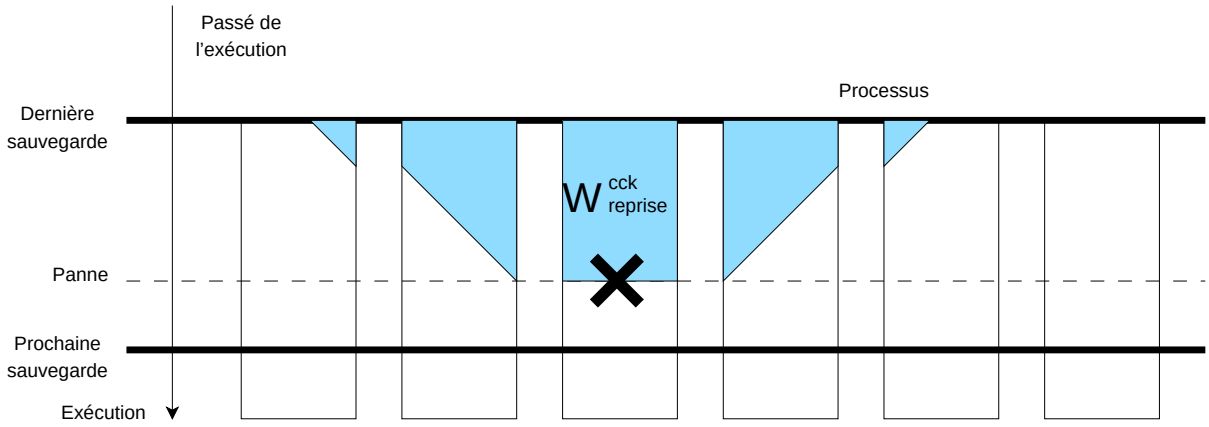


FIG. 4.10 – Travail nécessaire à la reprise avec le protocole CCK

4.3.8 Comparaison avec le protocole standard

Le protocole CCK induit un coût supplémentaire au redémarrage pour calculer l'ensemble de tâches à ré-exécuter. Mais le point fort de CCK est de réduire cet ensemble de tâches à ré-exécuter qui est la partie la plus coûteuse de la reprise. Cet ensemble correspond à $\mathcal{W}^{cck}_{reprise}$ et est illustré à la figure 4.10. On peut le comparer avec le travail $\mathcal{W}^{std}_{reprise}$ à ré-exécuter pour la reprise globale du protocole standard de la figure 4.11.

Ces deux figures montrent bien l'avantage de la reprise partielle du protocole CCK sur la reprise globale du protocole standard. Le reprise globale est très coûteuse, en particulier lorsque N, le nombre de processus, est très grand (on rappelle que $\mathcal{W}^{std}_{reprise} = O(N\tau)$). Le reprise partielle de CCK se comporte mieux quand N est grand car ce coût dépend essentiellement du nombre de processus défaillants. ($\mathcal{W}^{cck}_{reprise} = O((N_{défaillants} + \varepsilon)\tau^{cck})$).

De plus dans le cas du protocole CCK, la reprise est partielle : seul certains processus «voisins» des processus défaillants participent à la reprise. L'ensemble des processus, du fait qu'ils ne soient pas tous occupés à la reprise, peuvent être utilisés pour redistribuer la reprise. Typiquement, si suffisament de parallélisme existe, le travail $\mathcal{W}^{cck}_{reprise}$ peut s'exécuter en temps $\mathcal{W}^{cck}_{reprise}/N$ avec N processus.

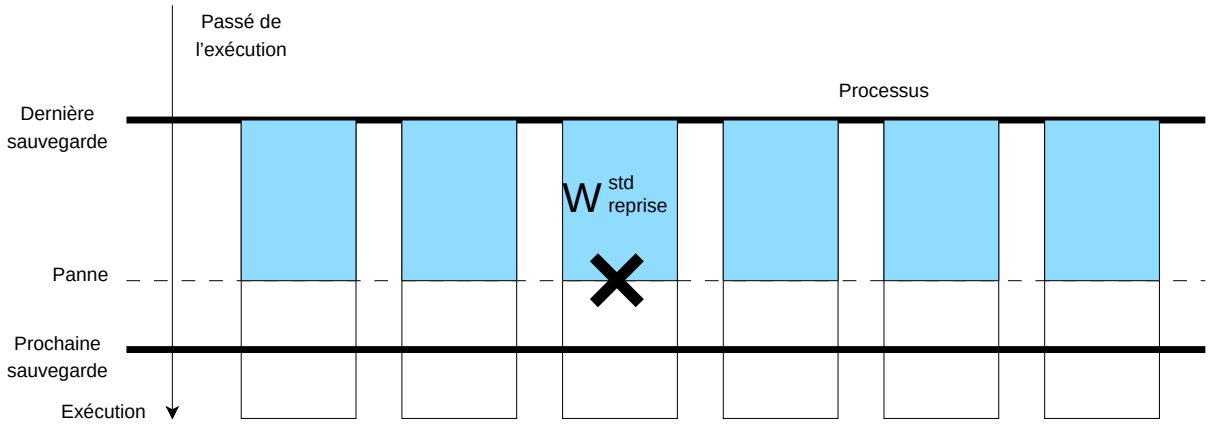


FIG. 4.11 – Travail nécessaire à la reprise avec le protocole standard

4.4 Améliorations possibles

Le modèle présenté dans les sections précédente comporte le protocole tel qu'il a été formalisé jusqu'à maintenant. Bien que le protocole CCK dispose de nombreuses améliorations par rapport au protocole standard, il peut encore être optimisé. Cette section a pour but de proposer plusieurs améliorations qui pourrait s'appliquer au protocole CCK tel qu'il a été présenté.

4.4.1 Sauvegarde non bloquante

Un des points négatif des protocoles de sauvegarde/reprise coordonnées est le coût induit par la coordination. Bien que cette coordination soit indispensable (si on veut construire un état cohérent à la sauvegarde), on peut tenter de la réduire au strict nécessaire. L'idée est d'éviter l'étape de signalisation de fin de sauvegarde (étape 4 du protocole à la figure 3.2). En pratique, cette étape sert à empêcher que les messages envoyés par les processus qui ont terminé la sauvegarde de leur état n'affectent l'état d'un processus qui n'a pas fini de sauvegarder. Pour cela, plusieurs méthodes sont possibles :

- On peut retarder la délivrance de ces messages jusqu'à la fin de la sauvegarde. Si un tel message est reçu, on le met dans une zone de mémoire tampon qui ne sera lu qu'après la fin de la sauvegarde. Cependant cela induit un surcoût sur les communications.
- On peut également créer un clone du processus grâce à un appel système `fork`. Le processus cloné a tout le temps de sauvegarder son état, les nouveaux messages n'affecteront que le processus original. Il faut toujours retarder la délivrance des messages cependant on peut espérer que ce soit pendant une durée plus faible car le `fork` effectue une copie paresseuse de la mémoire [4](copy-on-write).

4.4.2 Prise en compte des calculs déjà effectués

On propose ici une optimisation qui permet de réduire le nombre de tâches à ré-exécuter. Cette optimisation consiste à tenir compte du travail déjà effectué sur les processus non défaillants depuis la dernière sauvegarde.

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 lu par une tâche qui n'a pas encore été exécutée. Si une telle donnée est dans le graphe des tâches à ré-exécuter, elle peut être utilisée pour éliminer des tâches de calcul de l'ensemble des tâches à ré-exécuter.

Sur la figure 4.12, on peut voir l'ensemble des tâches à ré-exécuter pour reprendre le calcul en prenant en compte les données présentes en mémoire. Ces données disponibles sont celles dont la valeur va encore être utilisée par une tâches et donc elle n'a pas encore été détruite.

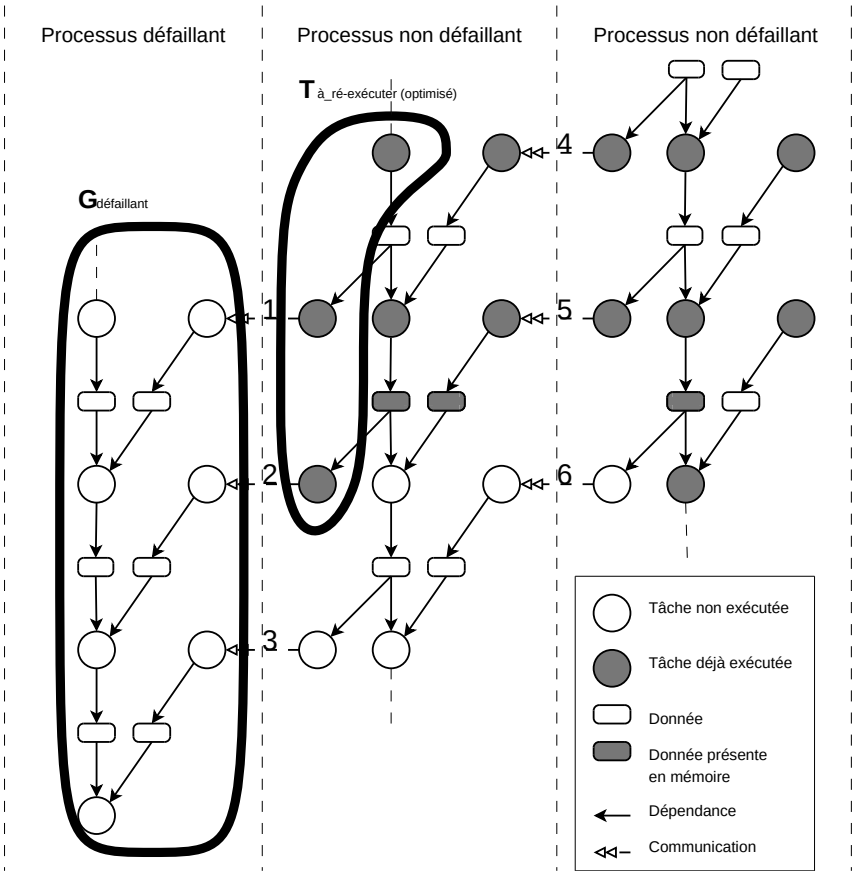


FIG. 4.12 – L'ensemble des tâches à ré-exécuter en prenant en compte les données disponibles en mémoire

4.4.3 Réordonnancement local

Cette optimisation est destinée à améliorer le temps nécessaire à la reprise avec le protocole CCK. Lors d'un redémarrage avec CCK, certaines tâches sont ré-exécutées de manière à pouvoir ré-émettre les communications perdues.

L'idée est alors d'essayer d'exécuter les tâches de communications au plus tôt (sans perturber le parallélisme de l'application) en opérant un réordonnancement local de celles-ci sur ce processus. Ainsi, lorsqu'une sauvegarde aura lieu, les communications déjà effectuées seront sauvegardées dans l'état des processus, et en cas de panne postérieure, elles n'interviendront pas dans les dépendances.

Un programme de simulation a été réalisé pour donner une idée de l'effet d'un tel réordonnancement. Le résultat des simulations est présenté au chapitre suivant.

4.5 Conclusion

Ce chapitre a décrit en détail le protocole CCK. Les algorithmes ont été expliqués et prouvé, les coûts ont été déterminés. Le protocole CCK a été comparé du point de vue théorique au protocole standard décrit au chapitre 3. Les améliorations apportées à CCK le rende plus performant que le protocole standard, aussi bien pour l'étape de sauvegarde (la coordination est plus efficace) que pour l'étape de reprise (le nombre de tâches à ré-exécuter est plus faible).

Le chapitre suivant présente l'état de l'implémentation qui a été réalisée pour le protocole CCK. Ensuite, on donne les résultats de plusieurs expériences qui ont été effectuées.

Chapitre 5

Expérimentations

Ce chapitre a pour objectif de tester notre modèle et son implémentation. Les expériences effectuées portent sur l'évaluation des performances du protocole CCK.

L'organisation de ce chapitre est la suivante : l'état de l'implémentation réalisée est présenté puis la méthodologie et les conditions d'expérimentations. Ensuite, il est fait état des expériences et de leurs résultats pour l'étape de sauvegarde et de reprise.

5.1 État de l'implémentation

Une partie de l'implémentation pour le protocole CCK a été réalisée. De nombreuses difficultés ont été rencontrées mais elles ne seront pas toutes détaillées. Cette section décrit l'état actuel de l'implémentation du protocole ainsi qu'une brève description de l'organisation du code.

5.1.1 Sauvegarde coordonnée

L'intégralité de la partie sauvegarde, telle qu'elle a été décrite dans ce chapitre, a été implémentée et est fonctionnelle.

Deux objets ont été créés :

- `CCKCheckpointCoordinator` : cet objet possède son propre thread qui exécute les différentes étapes du protocole de sauvegarde du coordinateur. Ce thread est *réveillé* périodiquement par une horloge. Pour le moment, cet objet est instancié sur le processus de calcul 0 mais il est prévu de le déplacer sur un processus dédié.
- `CCKCheckpointController` : cet objet est instancié sur chaque processus de calcul. Il possède également son propre thread qui exécute l'algorithme de sauvegarde du protocole CCK des processus de calcul. Ce thread est réveillé par l'arrivée d'un message INIT. Cet objet a la charge d'arrêter et de reprendre des calculs et de vider les canaux de communication.

5.1.2 Reprise

La partie reprise du protocole CCK est implémentée partiellement et n'est pas encore entièrement fonctionnelle. Les algorithmes de calcul de dépendances sont implémentés mais de manière centralisés. Ils ont permis de réaliser les simulations présentées dans ce chapitre. La partie concernant les synchronisations et les communications collectives sont encore absentes.

L'organisation est similaire à celle de la sauvegarde. On utilise également deux objets

- `CCKRestartCoordinator` pour le processus coordinateur ;
- `CCKRestartController` pour chaque processus de calcul.

Ces objets sont également dotés d'un thread qui est réveillé par le début d'une phase de redémarrage.

5.1.3 Validation de l'implémentation

Il a été nécessaire de valider l'implémentation de l'étape de sauvegarde. En particulier, il est important de vérifier que l'état sauvegardé fournit bien un état global cohérent qui permet de redémarrer. Une reprise globale telle que celle détaillée au chapitre 3 a été implémentée. Elle a permis de réaliser des reprises à partir des états sauvegardés et même plusieurs reprises avec des pannes successives.

5.2 Méthodologie d'expérimentation

5.2.1 Application de test : Jacobi2D

Pour réaliser les expérimentations, une application parallèle itérative est utilisée. L'application Jacobi2D est la version parallélisée de la méthode de résolution de Jacobi appliquée à une surface (2D). Cette application présente un schéma itératif et une proportion de communications importante par rapport aux calculs, ce qui caractérise bien le domaine d'application qui nous intéresse : les simulations de phénomènes physiques.

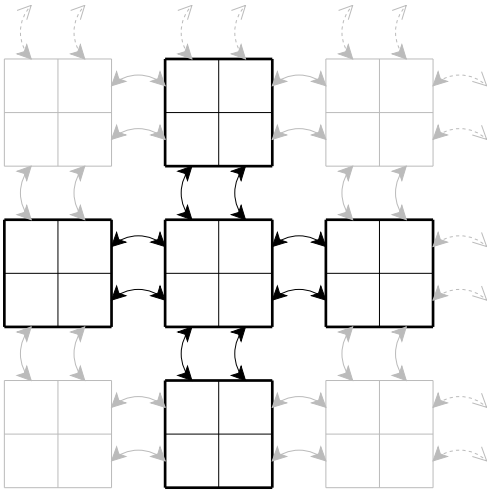


FIG. 5.1 – Exemple de découpage de domaines pour une application itérative. Chaque processus se voit attribuer 4 cellules. Les flêches indiquent les communications.

La figure 5.1 montre comment peut être découpé le domaine d'une application itérative. Les cellules de base sont regroupées par 4 sur un même processeur. L'état d'une cellule à un instant t dépend de son état et de l'état de ses voisins à l'instant $t-1$, donc passer de l'étape t à l'étape $t+1$ nécessite des échanges de données. Pour les cellules qui sont sur le même processeur, l'échange est implicite (c'est un accès à la mémoire). La figure montre les communications induites par ces échanges de données entre les processeurs. Dans ce cas, les dépendances directes entre les processus restent limitées à ses quatre voisins. Dans la suite, on appelle taille de cellule la taille des données à communiquer pour une cellule.

L'exécution est alors considérée comme une itération d'étapes de calcul et d'étapes de communications. L'exécution est alors déroulée à l'avance sur plusieurs itérations pour permettre à l'ordonnanceur de recouvrir les étapes de communications par du calcul.

5.2.2 Paramètres d'évaluation

Sauvegarde

L'évaluation de la sauvegarde du protocole CCK se fait en l'absence de panne. Les valeurs mesurées sont :

– le temps d'une étape de sauvegarde qui comprend le temps de coordination et le temps de la sauvegarde ;

– le temps d'exécution d'une instance de notre application. Ce temps d'exécution sera également comparé au temps d'exécution de la même instance sans le protocole CCK.

Les paramètres suivants sont étudiés :
– la période de sauvegarde τ ;
– le nombre de processeurs p ;
– le nombre s de supports stables (serveurs de sauvegarde) utilisés durant l'exécution.

Reprise

L'évaluation de la reprise du protocole CCK se fait en présence d'une ou plusieurs pannes. La valeur mesurée est le nombre de tâches à ré-exécuter. Ce nombre de tâches sera comparé au nombre de tâches à ré-exécuter dans le cas d'un redémarrage global en utilisant le protocole standard.

Les paramètres étudiés sont les suivants :
– la forme du graphe de flot de données (le rapport nombre d'itérations sur largeur du graphe) ;
– la date de la panne par rapport à la date de la dernière sauvegarde.

5.2.3 Environnement d'expérimentation

Les expérimentations sont réalisées sur la grappe d'Orsay de la plateforme Grid'5000. Grid'5000 est une grille expérimentale qui regroupe des grappes d'ordinateurs géographiquement réparties sur le territoire français (Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse) et qui totalise actuellement 1089 nœuds bi-processeurs, soit un total de 2178 processeurs.

La grappe d'Orsay possède la configuration suivante :
– 342 nœuds de type IBM eServer 326m ;
– Processeur : AMD Opteron 246, 2.0 ou 2.4 GHz, 1 Mo de cache, x 2 (soit 684 processeurs au total) ;
– Mémoire : 2 Go ;
– Réseau : Gigabit Ethernet ;
– Disque dur : 80 Go / SATA ;
– Système d'exploitation : Linux 2.6

5.3 Coût de la sauvegarde

Le coût d'une sauvegarde et le temps d'une exécution avec sauvegardes dépendant de plusieurs paramètres : nombre de processus, nombre de serveurs de sauvegarde, taille du graphe, période de sauvegarde. Cette section présente différentes expériences destinées à mesurer l'influence de ces paramètres dans le cas d'une exécution sans panne.

5.3.1 Influence de la période de sauvegarde

La période de sauvegarde est un paramètre critique pour l'exécution du protocole CCK. En effet, une période trop faible entraînera de trop nombreuses coordinations qui ralentiront le calcul. En revanche, une période trop élevée engendrera des sauvegardes moins régulières et ainsi la quantité de calcul perdu en cas de redémarrage deviendra conséquente.

La figure 5.2 montre le temps total d'exécution de l'application pour un domaine de taille 32×64 déroulée sur 10 itérations et exécutée sur 32 processeurs en fonction de la fréquence des sauvegardes. On retrouve le temps d'exécution de référence (c'est-à-dire sans utilisation du protocole) pour la fréquence nulle.

On remarque la présence d'une fréquence seuil 0.1 (soit une période de 10 secondes) au-dessus de laquelle les performances sont fortement dégradées. Pour notre exemple, la fréquence seuil est relativement basse car la taille des sauvegardes est assez importante : à chaque coordination, chaque processus

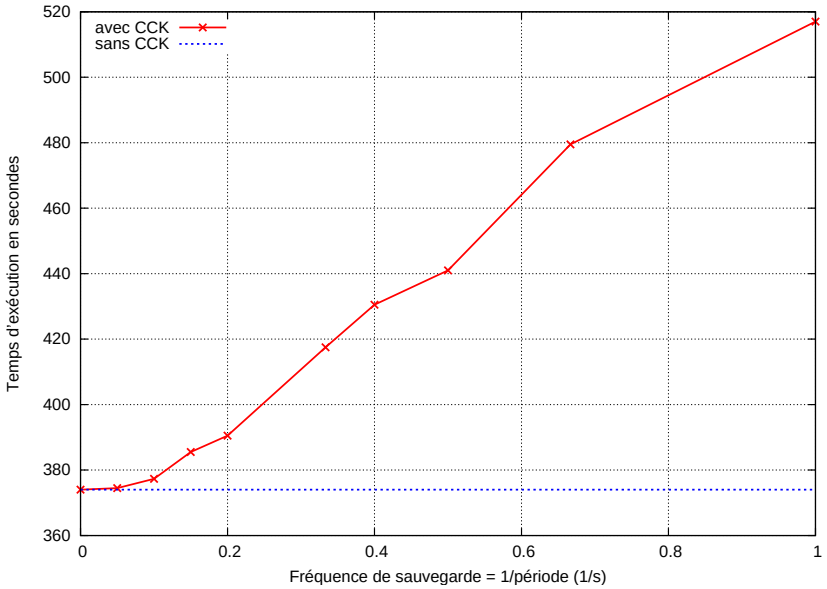


FIG. 5.2 – Influence de la fréquence de sauvegarde sur le temps total d'exécution (domaine de 32×64 sur 10 itérations sur 32 processeurs avec des cellules de 4 ko et 10 serveurs de sauvegarde)

sauvegarde environ 1,5 Mo de données. Le support stable de stockage constitue alors un point de congestion.

5.3.2 Influence du nombre de serveurs de sauvegarde

Les serveurs de sauvegarde sont un point de congestion durant l'exécution du protocole. On peut décider d'augmenter le nombre de serveurs de sauvegarde pour limiter la congestion et améliorer les performances. L'augmentation du nombre de serveurs de sauvegarde devrait donc diminuer le temps d'exécution de l'application.

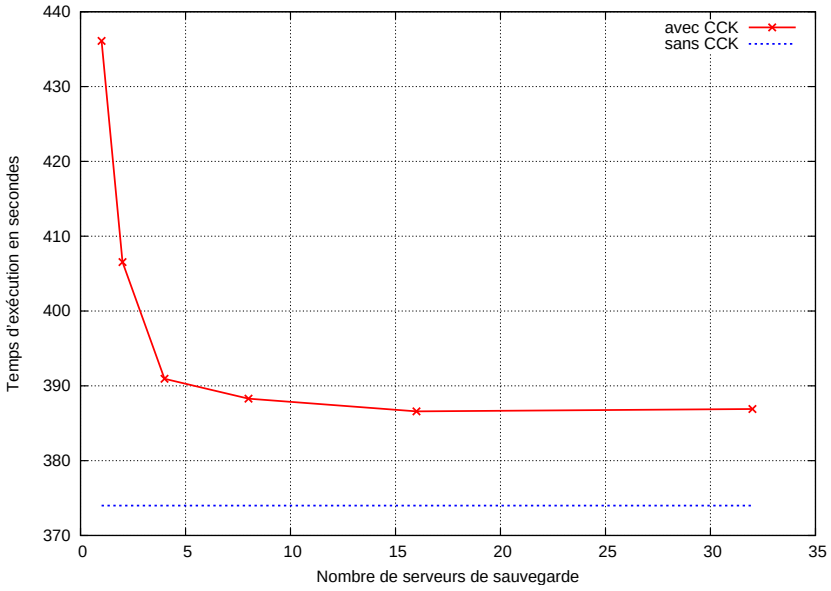


FIG. 5.3 – Influence du nombre de serveurs de sauvegarde sur le temps total d'exécution (domaine de 32×64 sur 10 itérations sur 32 processeurs avec des cellules de 4 ko et une période de sauvegarde de 5 secondes)

La figure 5.3 présente le temps d'exécution de l'application en fonction du nombre de serveurs de sauvegarde. L'instance choisie est Jacobi en deux dimensions avec un domaine de 32×64 sur 10 itérations

et des cellules de 4 ko. Elle a été exécutée sur 32 processeurs avec une période de sauvegarde de 5 secondes où le nombre de serveurs de sauvegarde varie entre 1 et 32. On constate qu'en dessous de 4 serveurs de sauvegarde les performances sont nettement détériorées. Chaque processus envoie une sauvegarde d'environ 1,5 Mo toutes les 5 secondes ; dans ces conditions, il faut donc au moins un serveur de sauvegarde pour 8 processus de calcul.

5.3.3 Influence du nombre de nœuds

Dans cette expérience, on mesure le temps d'une étape de sauvegarde en fonction du nombre de processus. Cette expérience permet de mettre en évidence les propriétés de passage à l'échelle du protocole CCK et en particulier son étape de coordination. Pour cette expérience, une attention toute particulière est placée sur la période de sauvegarde qui doit être suffisamment élevée. Egalement, le nombre de serveurs de sauvegarde doit être suffisant pour éviter toute congestion.

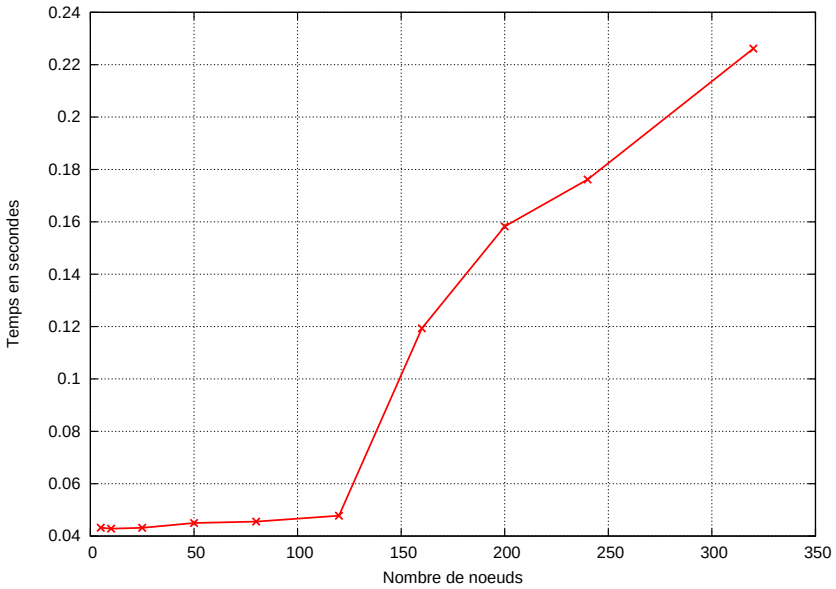


FIG. 5.4 – Coût d'une étape de sauvegarde en fonction du nombre de nœuds

La figure 5.4 montre le temps de coordination en fonction du nombre de nœuds. Le temps mesuré correspond au temps écoulé entre le début de l'arrêt et la fin du redémarrage des threads (cf figure 4.5). On constate ici une rupture dans la courbe pour un nombre de processeurs égal à 128. On n'a actuellement aucune explication mais des expérimentations supplémentaires sont prévues pour trouver une raison à ce phénomène. Il faut également remarquer que le temps de synchronisation pour 300 processus est d'environ 230 millisecondes, ce qui est relativement faible.

5.4 Coût du redémarrage

Cette partie de l'implémentation du redémarrage de CCK n'étant pas encore fonctionnelle, cette section présente des simulations de redémarrage en comptant le nombre de tâches à ré-exécuter selon différents cas de figure. On se place dans le cadre d'une application simple de type décomposition de domaine où chaque cellule possède deux voisins.

5.4.1 Nombre de tâches à rejouer

Les paramètres étudiés sont la forme du graphe et la date de la panne. On compte le nombre de tâches à ré-exécuter sur un voisin du processus défaillant en comparaison avec la reprise globale du

protocole standard. Les tâches à ré-exécuter sont les tâches qui ont déjà été exécutées mais qui vont être ré-exécutées pour fournir au processus défaillant les communications qui lui sont nécessaires.

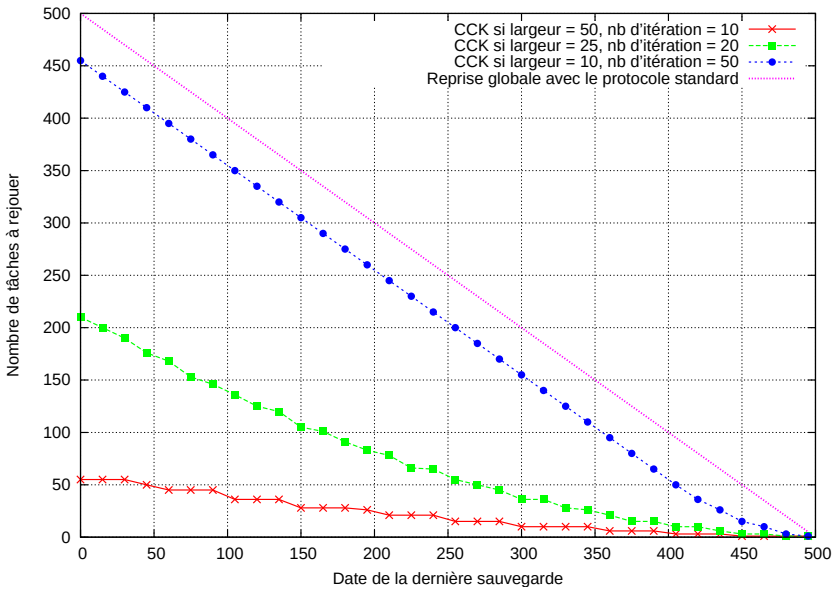


FIG. 5.5 – Nombre de tâches à ré-exécuter sur un voisin du processus défaillant lors de la reprise en fonction de la date de la dernière sauvegarde pour la reprise globale en utilisant pour le protocole CCK selon différentes formes de graphes

Notons que le cas traité est l'étude du nombre de tâches à rejouer sur un voisin direct au processus défaillant ce qui est l'un des pires cas avec celui du processus défaillant lui-même (cf 4.3.7). La forme du graphe influence énormement cette quantité de tâches à ré-exécuter : un graphe « large » (peu d'itérations par rapport à la taille du domaine) est plus favorable car il y a moins de dépendances entre les tâches (les dépendances se font entre deux itérations). Dans tous les cas, le nombre de tâches à ré-exécuter reste inférieur au cas de la reprise globale du protocole standard.

5.4.2 Influence du réordonnancement local

Cette simulation veut montrer l'influence de l'optimisation proposée pour CCK qui est de réordonnancer localement les tâches pour favoriser les tâches d'émission des communications.

Ce programme simule le comportement d'une application simple de type décomposition de domaine à une dimension avec ou sans réordonnancement local. La date de la dernière sauvegarde et la date de la panne sont fixées et on obtient en retour un décompte des tâches à ré-exécuter en tenant compte des dépendances entre les tâches pour l'application considérée.

Le résultat de cette simulation est donné à la figure 5.6. Cette figure montre le nombre de tâches à ré-exécuter sur un voisin direct du processus défaillant en fonction de la date de la dernière sauvegarde et de la date de la panne. Cette simulation ne prend pas en compte les tâches déjà exécutées sur le processus voisin du processus défaillant.

Cette simulation met en évidence deux phénomènes :

– Tout d'abord, si la panne a lieu avant les premières sauvegardes, on s'aperçoit que le nombre de tâches à ré-exécuter est beaucoup plus important. En effet, grâce à cet ordonnancement les tâches exécutées en premier sont celles qui permettent de générer des communications et si elles ne sont pas sauvegardées, elles devront être rejouées. Dans le cas extrême, les tâches exécutées sont nécessaires au calcul interne et pourront être réutilisés.

– De plus, on voit apparaître un seuil au-delà duquel, si une panne se produit, il n'y a rien à ré-exécuter sur les voisins car toutes les communications ont été sauvegardées. Sur la figure, ce phénomène apparaît après la deuxième sauvegarde.

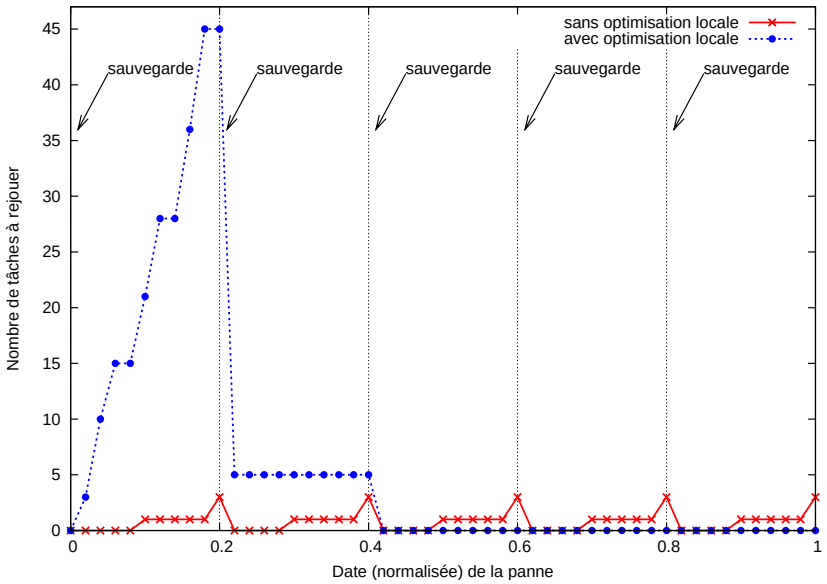


FIG. 5.6 – Simulation du nombre de tâches à ré-exécuter en fonction de la date de la panne avec et sans optimisation de réordonnancement local (la largeur du graphe utilisé est 50 et le nombre d'itération est 10)

5.5 Conclusion

Les expériences réalisées montrent les performances de notre protocole CCK. Les expériences réalisées mettent en évidence les paramètres important pour l'étape de sauvegarde. Ces paramètres sont : la période de sauvegarde, le nombre de serveurs de sauvegarde et le nombre de processus.

Pour l'étape de reprise, seules des simulations ont été réalisées. Elles permettent d'avoir un ordre d'idée sur le nombre de tâches à ré-exécuter pour la reprise CCK par rapport à la reprise globale. Elles mettent également en évidence un paramètre important : la forme du graphe. Pour notre simulation, cette forme permet de caractériser les dépendances internes des tâches de l'application ; un graphe « large » présente peu de dépendances, un graphe « long » engendre beaucoup de dépendances.

De nombreuses expériences sont également en prévision. Tout d'abord, la fin de l'implémentation de la reprise CCK permettra de le tester sur de vraies applications et également de tester l'optimisation de réordonnancement local.

Une seconde étape d'expérimentations sera de comparer le protocole CCK au protocole TIC. Le protocole TIC (Theft-Induced Checkpointing) [17] est un protocole de tolérance aux pannes basé sur la sauvegarde induite par les communications qui a été implanté dans KAAPI. Une série d'expériences pourra mesurer et comparer les performances des deux protocoles sur différentes catégories d'applications.

Chapitre 6

Conclusion

Dans ce travail, un protocole standard de tolérance aux pannes par sauvegarde/reprise coordonnée est étudié. Plusieurs améliorations de ce protocole standard sont proposées ce qui a amené à la conception d'un nouveau protocole de tolérance aux pannes adapté à l'intergiciel KAAPI : le protocole CCK (Coordinated Checkpointing in KAAPI). Elles portent à la fois sur l'étape de coordination lors de la sauvegarde et sur la phase de reprise. Les améliorations proposées par ce protocole reposent essentiellement sur l'exploitation de la représentation abstraite sous forme d'un graphe de flot de données utilisé par KAAPI. Ce modèle offre une représentation du futur de l'exécution.

Le protocole CCK est détaillé : l'algorithme est expliqué et prouvé, une analyse de coût est effectuée. Une implémentation partielle de ce protocole a été réalisée et des expériences montrent ses performances : l'étape de coordination est relativement rapide même sur plusieurs centaines de processeurs, l'algorithme de reprise qui est une originalité du protocole CCK permet de réduire la quantité de travail perdu en cas de pannes.

Par cette étude, le modèle de flot de données montre qu'il est très intéressant pour représenter l'exécution d'une application. C'est un modèle adapté pour la sauvegarde de l'état d'un processus (en particulier en milieu hétérogène) ; il permet d'avoir un contrôle et reflète l'évolution dynamique de l'exécution de l'application.

Perspectives

Les travaux réalisés et les résultats obtenus offrent plusieurs perspectives intéressantes. Cette partie présente et développe quelques idées qui pourraient faire suite et compléter l'étude du protocole CCK.

Protocole adaptatif TIC/CCK

TIC (Theft-Induced Checkpointing) [17] est un autre protocole de tolérance aux pannes pour KAAPI qui est un de type sauvegarde induite par les communications. Ce protocole a été conçu pour les applications ordonnancées par vol de travail.

Les protocoles de tolérance aux pannes TIC et CCK ont chacun leur domaine d'application. Tandis que le protocole TIC est efficace avec un ordonnancement par vol de travail pour des applications série-parallèle (où en pratique les vols sont faibles), le protocole CCK se révèle être adapté aux applications qui communiquent beaucoup tel que les applications itératives de simulation numérique.

L'idée est alors de créer un protocole hybride TIC/CCK qui s'adapte automatiquement à l'état de l'application. Un tel protocole serait intéressant en pratique pour les applications utilisant un ordonnancement par vol de travail. En effet, pendant une grande partie de l'exécution le nombre de vols effectués est faible. Cependant, à la fin de l'exécution le nombre de vols devient très important et dans ce cas, le surcoût lié à l'utilisation du protocole TIC devient non négligeable. Dans ce cas, l'utilisation du protocole CCK permet d'éclipser cet inconvénient.

Plusieurs problèmes sont à résoudre pour réaliser ce protocole adaptatif.

– Tout d'abord, cela nécessite un module de surveillance de l'application. Un tel module a pour rôle de contrôler régulièrement plusieurs variables caractérisant l'exécution de l'application. Les variables à surveiller peuvent être nombreuses. Par exemple, le nombre de requêtes de vols ou le nombre de messages échangés entre les processus sont des variables importantes pour ce protocole adaptatif. D'autres variables à surveiller sont à déterminer.
– Ensuite, il faut déterminer une série de règles qui permettent de réagir aux événements générés par le module de surveillance. Cet ensemble de règles permet d'automatiser le processus de décision. Ces règles peuvent être déterminées soit en manière théorique (en effectuant notament des analyses de coût), soit de manière expérimentale. Une règle simple peut être : si le nombre de requêtes de vols est supérieur à C, alors utiliser le protocole CCK. Il convient d'étudier des algorithmes de contrôle adaptés.
– Un point important est de pouvoir garantir la cohérence de l'état global et la possibilité de reprise quelle que soit la variante du protocole utilisée. Les sauvegardes réalisées par le protocole CCK à une étape donnée forment un état global cohérent. Les sauvegardes réalisées par le protocole TIC sont indépendantes et nécessitent la reconstruction d'un état global cohérent à la reprise. Le passage d'un protocole à l'autre peut donc nécessiter une étape intermédiaire pour garantir cela.

Redémarrage sur moins de ressources

Lorsqu'une panne se produit, on n'a pas toujours une machine supplémentaire qui permet de redémarrer le processus défaillant. Ainsi, une direction intéressante serait d'étendre le protocole CCK pour lui donner la possibilité de faire un redémarrage avec moins de processus.

Après une panne, l'algorithme de reprise de CCK détermine l'ensemble du calcul à ré-exécuter pour permettre le redémarrage. Ce calcul se présente sous forme d'un graphe de flot de données et il peut être partitionné pour répartir l'ensemble des calculs sur moins de processus.

On peut également envisager cette solution pour équilibrer la charge lors de l'ajout d'une grande quantité de ressources supplémentaires.

Bibliographie

[1] KAAPI : Kernel for adaptative, asynchronous parallel and interactive programming. http ://gforge.inria.fr/projects/kaapi/.

[2] A. Avizienis. Fault-tolerant systems. *IEEE Trans. Computers*, 25(12) :1304–1312, 1976.

[3] A. Avizienis, J.-C. Laprie, and B. Randell. Dependability and its threats - a taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, 2004.

[4] M. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[5] R. Baldoni. A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, page 68. IEEE Computer Society, 1997.

[6] F. Baude, D. Caromel, C. Delbé, and L. Henrio. A hybrid message logging-cic protocol for constrained checkpointability. In *Euro-Par*, pages 644–653, 2005.

[7] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. pages 133–147, 1997.

[8] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. Mpich-v : Toward a scalable fault tolerant mpi for volatile nodes. In *SuperComputing*, Baltimore, USA, 2002.

[9] A. Bouteiller, F. Cappello, T. Hérault, P. Lemarinier, G. Krawezik, and F. Magniette. Mpich-v2 : a fault tolerant mpi for volatile nodes based on the pessimistic sender based message logging. In *SuperComputing*, Phoenix, USA, 2003.

[10] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. In *In proceedings of The 2003 IEEE International Conference on Cluster Computing*, Honk Hong,China, 2003.

[11] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, 1996.

[12] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique*. Dunod, 1992.

[13] L. Debreu and E. Blayo. On the schwarz alternating method for oceanic models on parallel computers. *J. Comput. Phys.*, 141(2) :93–111, 1998.

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

[15] L. V. Kalé G. Zheng, L. Shi. Ftc-charm++ : An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September 2004.

[16] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1 : On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, pages 88–95, Paris, France, October 1998.

[17] S. Jafar. *Programmation des systèmes parallèles distribuées : tolérance aux pannes, résilience et adaptabilité*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, Juin 2006.

[18] S. Jafar, T. Gautier, A. W. Krings, and J.-L. Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In *Euro-Par*, pages 675–684, 2005.

[19] S. Jafar, A. W. Krings, T. Gautier, and J.-L. Roch. Theft-induced checkpointing for reconfigurable dataflow applications. In IEEE, editor, *IEEE Electro/Information Technology Conference , (EIT 2005)*, Lincoln, Nebraska, May 2005. This paper received the EIT'05 Best Paper Award.

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

[21] L. Kal, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. Namd2 : greater scalability for parallel molecular dynamics. *J. Comput. Phys.*, 151(1) :283–312, 1999.

[22] L. V. Kale and S. Krishnan. *Parallel Programming using C++*, chapter Charm++ : Parallel Programming with Message-Driven Objects, pages 175–213. MIT Press, 1996.

[23] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning : Applications in VLSI domain. Technical report, 1997.

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

[25] K. Marzullo L. Alvisi. Message logging : Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2) :149–159, 1998.

[26] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25, May 1987.

[27] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, 1978.

[28] J. C. Laprie. *ARAGO 15 : Concepts de base de la tolérance aux fautes*. MASSON, Paris, 1994.

[29] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[30] MOAIS. Inria project : Multi-programmation et ordonnancement sur ressources distribuées pour les applications interactives de simulation, 2006. http ://www.inria.fr/recherche/equipes_ur/moais.fr.html.

[31] M.Wiesmann, F. Pedonne, and A .Schipper. A systematic classification of replited database protocols based on atomic broadcast. *In Proceedings of the 3th European Research Seminar on Advances in Distributed Systems(ERSADS99)*, pages 351–360, 1999.

[32] Elnozahy E. N. *Manetho : Fault-Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*. Phd thesis, Rice University, October 1993.

[33] R.H.B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2) :165–169, 1995.

[34] F. Pellegrini and J. Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical Report 1038-96, 1996.

[35] L. Pigeon. Conception d'une bibliothèque pour les opérations de communication collective pour le langage de haut niveau ATHAPASCAN. Master's thesis.

[36] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, 1975.

[37] R. Revire, F. Zara, and T. Gautier. Efficient and easy parallel implementation of large numerical simulation. In Springer, editor, *Proceedings of ParSim03 of EuroPVM/MPI03*, pages 663–666, Venice, Italy, 2003.

[38] J. L. Roch, T. Gautier, and R. Revire. Athapascan : Api for asynchronous parallel programming. Technical Report RT-0276, `www-id.imag.fr/software/ath1`, Projet APACHE, INRIA, February 2003.

[39] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems (2nd ed.) : design and evaluation*. Digital Press, Newton, MA, USA, 1992.

[40] J. Silc, B. Robic, and T. Ungerer. Asynchrony in parallel computing : from dataflow to multithreading. pages 1–33, 2001.

[41] G. Stellner. CoCheck : Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[42] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3) :204–226, 1985.

[43] Y. Tamir and C. Sequin. Error recovery in multicomputers using global checkpoints. In *Proc. of 1984 International Conference on Parallel Processing*, pages 32–41, August 1984.

[44] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley and Sons, New York, 2001.

[45] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Proceedings of SuperComputing2000*, November 2000.

[46] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin : Simple and efficient Java-based grid programming. *Scalable Computing : Practice and Experience*, 6(3) :19–32, September 2005.

[47] Y. Wang, P. Chung, I. Lin, and W.K.Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5) :546–554, May 1995.

[48] F. Zara, F. Faure, and J.-M. Vincent. Physical cloth simulation on a pc cluster. In X. Pueyo D. Bartz and E. Reinhard, editors, *Fourth Eurographics Workshop on Parallel Graphics and Visualization 2002*, Blaubeuren, Germany, September 2002.