Un protocole de sauvegarde / reprise coordonné pour les applications à flot de données reconfigurables

Xavier Besseron — Laurent Pigeon[1] — Thierry Gautier — Samir Jafar[2]

INRIA – ID (équipe MOAIS), Grenoble, France

{Xavier.Besseron, Laurent.Pigeon, Thierry.Gautier, Samir.Jafar}@imag.fr

RÉSUMÉ. *La probabilité d'apparition d'une défaillance durant l'exécution d'une application parallèle de longue durée sur un grand nombre de processeurs est importante. Dans cet article, nous considérons l'étude d'un nouveau protocole de sauvegarde/reprise basé sur la construction coordonnée d'un état global cohérent, et dont la reprise après panne d'un processus ne nécessite qu'un redémarrage partiel de certains autres processus. Ce protocole repose sur l'existence d'une représentation abstraite de l'exécution.*

ABSTRACT. *Fault tolerance protocols play an important role in today long runtime scientific parallel applications because the probability of failure may be important due to the number of unreliable components involved during simulation. In this paper we present our approach and preliminary results about a new checkpoint/recovery protocol based on a coordinated scheme. One feature of this protocol is that recovery after a fault only requires a partial restart of other processes. This protocol is highly coupled to the availability of an abstract representation of the execution.*

MOTS-CLÉS : *grille, tolérance aux pannes, calcul parallèle, flot de données*

KEYWORDS: *grid, fault tolerance, parallel computing, dataflow graph*

1. Cet auteur est financé par l'Institut Français du Pétrole
2. Cet auteur est financé par une subvention du gouvernement Syrien

e soumission à *TSI, numéro spécial Renpar*, le 16 décembre 2006

1. Introduction

Depuis quelques années, la tolérance aux pannes est étudiée dans le contexte des applications parallèles à hautes performances qui permettent de simuler des phénomènes complexes *via* l'utilisation d'une architecture de type grappe (Jafar *et al.*, 2005b; Bouteiller *et al.*, 2003). De par le nombre de composants non fiables prenant part à la simulation, l'apparition de fautes matérielles n'est pas un événement exceptionnel : le système ou l'intergiciel doit pouvoir offrir des protocoles de tolérance aux pannes afin de masquer les défaillances. De plus, des réservations exclusives de ressources de calcul sur de longues durées rentrent en conflit avec les politiques de réservation visant à garantir l'équité entre utilisateurs sur des périodes courtes. Ce sujet a été largement étudié dans le contexte des systèmes et intergiciels distribués (Elnozahy *et al.*, 2002; Zheng *et al.*, 2004). Un nouvel axe de recherche vise à rendre les protocoles de tolérance aux pannes performants pour le calcul à hautes performances sur des architectures de très grande taille. Des propositions récentes étudient le comportement à l'exécution des applications afin de spécialiser ou d'étendre les protocoles connus (Jafar *et al.*, 2005a; Bouteiller *et al.*, 2003; Baude *et al.*, 2005).

L'objectif est de réduire le surcoût dû à l'utilisation d'un protocole de tolérance lors d'une exécution avec ou sans panne. Notre contribution s'inscrit dans ce contexte : la spécialisation du protocole de tolérance aux pannes est réalisée au niveau d'une représentation abstraite de l'exécution qui permet des optimisations importantes à l'exécution. Nous intégrons nos travaux à l'environnement d'exécution KAAPI (Jafar *et al.*, 2005b; Jafar *et al.*, 2005a), où cette représentation fut initialement conçue pour l'intégration de différents algorithmes d'ordonnancement s'adaptant au type d'application et permettant une virtualisation du nombre de ressources. Dans (Jafar *et al.*, 2005b), nous avons montré que cette représentation abstraite est bien adaptée pour la définition du point de sauvegarde d'un processus local. Dans ce papier, cette représentation abstraite est utilisée pour spécialiser un protocole de tolérance aux pannes pour l'exécution de simulations itératives qui communiquent beaucoup et dont le temps d'exécution est important.

Les protocoles coordonnés de sauvegarde/reprise sont adaptés à ces applications parallèles de grande échelle car ils n'ajoutent pas ou peu de surcoûts comparés aux protocoles de journalisation. Des expériences ont été menées en ce sens et démontrent que ces protocoles sont performants jusqu'à plusieurs milliers de processeurs (Zheng *et al.*, 2004; Bouteiller *et al.*, 2003). En cas de défaillance, tous les processeurs redémarrent depuis leur dernier point de sauvegarde, même ceux n'ayant pas été perturbés par la panne. Les réponses à chacune des trois questions suivantes permettent d'améliorer les performances de ces protocoles coordonnés de sauvegarde/reprise.

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 travail perdu en cas de panne ?

Nous apportons une réponse à (1) en proposant un algorithme de coordination inspiré de (Koo *et al.*, 1987) qui utilise les dépendances entre les processus pour réduire le nombre de messages échangés. Ces dépendances entre les processus sont déduites dynamiquement de la représentation abstraite de notre application.

Dans (Zheng *et al.*, 2004; Bouteiller *et al.*, 2003), la solution pour répondre à (2) est la suivante : chaque processeur conserve une copie locale de ses sauvegardes et envoie une autre copie soit à une zone de stockage stable (Bouteiller *et al.*, 2003) soit à un nombre fixé de processeurs voisins (Zheng *et al.*, 2004). En suivant cette approche, tous les processeurs, à l'exception de celui défaillant, redémarrent depuis la copie locale de leur dernier point de sauvegarde.

Notre contribution propose essentiellement une solution pour (3). Grâce à la représentation abstraite de l'exécution des applications écrites avec KAAPI, il est possible de calculer le *calcul strictement nécessaire* qui est l'ensemble des tâches de calcul qu'un processus doit rejouer pour envoyer les communications perdues vers un processus défaillant. Cela permet de réduire le travail total nécessaire sur les processus encore vivants pour le redémarrage du processus défaillant. Si ce travail à ré-excécuter contient suffisamment de parallélisme, il peut être réparti sur tous les processus vivants ; alors notre solution à la question (3) permettra aussi de répondre à (2).

Le plan de l'article est le suivant. La section suivante propose un survol des travaux liés aux mécanismes de tolérance aux pannes pour les applications parallèles. La section 3 présente notre protocole coordonné de sauvegarde/reprise pour les applications KAAPI. Elle commence par la description de la représentation abstraite dans KAAPI puis nous définissons ce qu'est l'état local d'un processus. Ensuite, nous présentons le protocole ainsi que sa complexité. Enfin, nous présentons les résultats préliminaires obtenus sur des applications de calcul scientifique.

2. État de l'art

Dans cet article, nous traitons les applications parallèles s'exécutant sur des environnements de type grappe ou grille de calcul. Les applications visées sont des applications itératives de calcul scientifique qui simulent l'évolution des phénomènes physiques, telles que les applications de dynamique moléculaire (Kal *et al.*, 1999), de réalité virtuelle (Zara *et al.*, 2002) ou de décomposition de domaine (Revire *et al.*, 2003). Ces applications sont caractérisées par un temps d'exécution important dont le ratio temps de communication sur temps de calcul peut être grand.

Les pannes pouvant survenir durant l'exécution d'applications parallèles peuvent être très variées (Avizienis *et al.*, 2004) : on distingue les pannes franches (le système ne répond plus) des pannes byzantines (le système réagit de manière aléatoire) ; ou bien les pannes transitoires (isolées dans le temps) des pannes permanentes (persistantes jusqu'à réparation). Dans la suite, nous nous intéressons aux pannes franches ou des pannes pouvant être modélisées ou considérées comme des pannes franches.

Les protocoles de tolérance aux pannes sont classés en trois catégories (Elnozahy *et al.*, 2002) : ceux basés sur la duplication afin d'introduire une redondance des exécutions (Avizienis, 1976; Wiesmann *et al.*, 1999), les protocoles de journalisation des événements (Alvisi *et al.*, 1998; Strom *et al.*, 1985) et enfin, les protocoles basés sur une approche de type sauvegarde/reprise (Elnozahy *et al.*, 2002; Chandy *et al.*, 1985).

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. Le principal désavantage de cette méthode est qu'elle nécessite un nombre important de ressources : pour tolérer N pannes, il est nécessaire d'avoir $N + 1$ composants identiques. C'est pourquoi ces protocoles ne sont pas utilisés lorsque le critère de performance est le temps d'exécution (Elnozahy *et al.*, 2002; Zheng *et al.*, 2004; Bouteiller *et al.*, 2003).

Les protocoles de journalisation font l'hypothèse PWD (*PieceWise Deterministic assumption*) (Strom *et al.*, 1985), c'est-à-dire que l'état de l'application évolue selon des événements non déterministes. Entre ces événements, l'exécution de l'application est déterministe. Le principe de la journalisation est alors de sauvegarder l'historique de l'application en créant un journal des événements non déterministes. En cas de panne, le processus défaillant est capable de reconstruire son état d'avant la panne en rejouant les événements non déterministes inscrits dans son journal. Dans notre cas, les événements non déterministes sont les réceptions de messages. Les auteurs de (Bosilca *et al.*, 2002; Bouteiller *et al.*, 2003) montrent un surcoût important dans le cas d'applications qui communiquent beaucoup. Ces protocoles ne sont donc pas adaptés.

Les protocoles de sauvegarde/reprise ont pour objectif de réaliser une sauvegarde de l'état de l'application à un instant donné. 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. Le rôle du protocole de tolérance aux pannes par sauvegarde/reprise est de reconstruire un état global cohérent à partir de l'état potentiellement incohérent du système après une panne et des informations sauvegardées.

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 » l'envoi de ce message. C'est-à-dire que l'événement associé à l'envoi du message précède l'événement associé à la prise d'état du processus. (Elnozahy* et al.*, 2002; Chandy* et al.*, 1985).*

L'état global cohérent reconstruit n'est pas nécessairement un état de l'application avant la panne, il suffit qu'il représente un état de l'application qui aurait pu se produire durant une exécution sans panne.

Dans la figure 1 à droite, l'état global formé par les losanges sur chacune des lignes de temps des trois processus P_0, P_1 etP_2 n'est pas cohérent 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. Cette situation ne peut jamais se produire dans une exécution correcte et c'est pourquoi l'état global est incohérent. Le message m_1 est appelé un message **orphelin**.

Dans le schéma de gauche de la figure 1, l'état global contient certains messages (m_2 et m_3) envoyés mais pas reçus. Ces messages sont appelés messages **en transit** et ils font partie de l'état global du système (application). Les messages en transit n'introduisent aucune incohérence dans l'état. Cependant, la garantie de la délivrance des messages en transit doit être assurée, soit par l'hypothèse d'un réseau de communication fiable, soit par le protocole de sauvegarde/reprise lui-même.

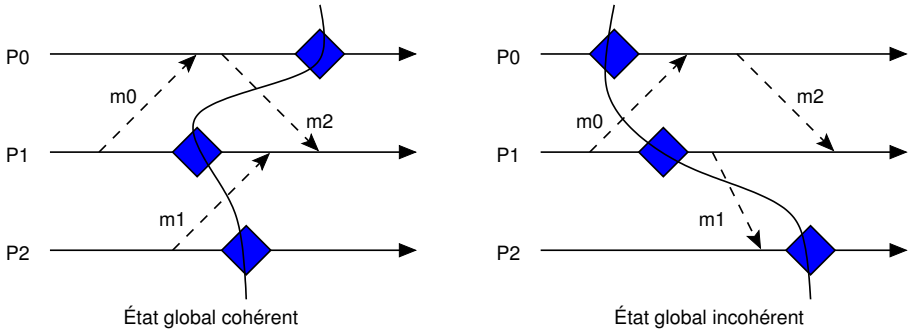

Figure 1. *Exemple d'état global cohérent et incohérent. Les axes horizontaux représentent le temps pour trois processus P_0, P_1 et P_2. Les flèches représentent des communications entre les processus. Les losanges représentent les événements associés à la capture de l'état local des processus.*

Les protocoles de sauvegarde/reprise se déclinent en trois catégories selon le mode de construction de l'état global cohérent (Elnozahy *et al.*, 2002) à partir des sauvegardes.

Pour les protocoles non-coordonnés, les processus effectuent leurs sauvegardes de manière indépendante. Il n'y a donc aucune garantie sur la cohérence de l'état global capturé. Ainsi, un effet domino peut se produire : l'état global cohérent représente, dans le pire cas, le calcul initial (Randell, 1975).

Les protocoles coordonnés synchronisent les processus pour effectuer la sauvegarde de leur état. Cette synchronisation assure la cohérence de l'état global sauvegardé. La reprise s'effectue à partir de la dernière sauvegarde. Ces protocoles sont basés sur les travaux de Chandy et Lamport (Chandy *et al.*, 1985). Ils font l'hypothèse de canaux de communication FIFO et utilisent des techniques pour vider les canaux de communication. Parmi ces protocoles, on peut distinguer les protocoles qui effectuent une sauvegarde coordonnée minimale pour laquelle la synchronisation n'est effectuée qu'entre certains processus (synchronisation nécessaire au regard des communications). Un tel protocole est présenté dans (Koo *et al.*, 1987).

Les protocoles de sauvegarde induite par les communications (*Communication-Induced Checkpointing*) (Baldoni, 1997) représentent une solution hybride entre les protocoles coordonnés et non-coordonnés. En plus des sauvegardes locales de chaque processus, des sauvegardes forcées sont effectuées à la réception de certains messages de manière à empêcher l'effet domino en cas de reprise.

Les protocoles de sauvegarde/reprise coordonnés ont l'avantage de posséder un faible surcoût vis-à-vis des communications (Zheng *et al.*, 2004; Bouteiller *et al.*, 2003). Néanmoins, ils engendrent un volume de communication lié à la taille de l'état sauvegardé. Ce coût peut-être amorti en choisissant la période entre deux sauvegardes.

3. Protocole de sauvegarde/reprise amélioré

L'idée du protocole CCK (*Coordinated Checkpointing in* KAAPI) est de construire, après la détection d'une défaillance, la quantité de travail minimale à rejouer afin de ré-envoyer les messages vers le processus défaillant. Grâce à la représentation abstraite des applications KAAPI, notre protocole nécessite une moindre quantité de calcul à ré-exécuter comparée aux protocoles coordonnés classiques et améliorés (Elnozahy *et al.*, 2002; Zheng *et al.*, 2004; Bouteiller *et al.*, 2003) pour lesquels tous les processus redémarrent de leur dernière sauvegarde. Cette section présente la méthode qui permet, grâce à l'exploitation de la représentation abstraite du calcul fournie par KAAPI, de réduire le nombre de tâches qui doivent être ré-exécutées. Nous commençons par décrire le modèle d'exécution et la représentation abstraite de l'exécution dans KAAPI. Puis nous présentons notre protocole coordonné CCK.

3.1. *Modèle d'application et hypothèses*

Les applications considérées ont un comportement déterministe, c'est-à-dire que le même calcul sur les mêmes données produisent le même résultat. Il n'y a pas d'effet de bord. De plus, on suppose que les canaux de communication sont FIFO : les messages sont délivrés dans l'ordre où ils ont été envoyés. Enfin, les pannes considérées sont les pannes franches.

3.2. *Modèle d'exécution et représentation abstraite de l'exécution*

KAAPI (`http://gforge.inria.fr/projects/kaapi/`) 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.

3.2.1. *Modèle de programmation*

Le modèle de programmation proposé par KAAPI (Jafar, 2006) est un modèle de programmation parallèle de haut niveau. Selon cette approche, le programmeur écrit

son programme en définissant explicitement le parallélisme potentiel de son application et ceci indépendamment de l'architecture matérielle cible. Pour cela, KAAPI est couplé à une interface de programmation de haut niveau : Athapascan (Galilée *et al.*, 1998; Roch *et al.*, 2003). Cette API permet de définir le parallélisme selon deux concepts simples : les **données partagées** et les **tâches**. Les données partagées sont des données en mémoire (virtuellement partagée) accessibles par les tâches. Les tâches constituent un ensemble indivisible d'instructions.

Les tâches déclarent des modes d'accès (lecture, écriture) sur les données partagées. Ceci permet au moteur exécutif KAAPI de calculer à l'exécution les contraintes de précédence entre les tâches. L'ensemble {tâches, données, précédence} décrivent un **graphe de flot de données** associé à l'exécution de l'application (Galilée, 1999; Roch *et al.*, 2003).

Le graphe de flot de données d'une application utilisant KAAPI est appelé **représentation abstraite** de l'application car cette représentation est causalement connectée à (l'exécution de) l'application (Jafar, 2006) : chaque exécution d'une tâche (instruction spécifique du langage Athapascan) est reportée par la création d'un nouveau sommet et d'arcs dans le graphe de flot de données et toute modification du graphe de flot de données est traduite en une modification de l'exécution de l'application.

La représentation abstraite par graphe de flot de données définie ci-dessus peut-être utilisée pour construire un état global d'une exécution distribuée (Jafar, 2006).

3.2.2. *Moteur d'exécution*

Le moteur d'exécution de KAAPI gère un ensemble de processeurs virtuels. Ces processeurs virtuels sont repliés sur des processeurs physiques. Chaque processeur virtuel exécute des portions du graphe de flot de données.

Le travail est réparti entre les processeurs virtuels par un algorithme de partitionnement de graphe.[1]

L'ordonnancement statique permet de partitionner le graphe de flot de données associé à un processeur virtuel en K sous-graphes qui seront distribués sur K processeurs virtuels. KAAPI peut utiliser les librairies SCOTCH (Pellegrini *et al.*, 1996) ou METIS (Karypis *et al.*, 1997) pour effectuer le partitionnement (Revire, 2004; Pigeon, 2007). 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.

La figure 2 montre un graphe de flot de données après l'étape d'ordonnancement statique et sa distribution sur trois processus.

1. KAAPI permet aussi de répartir le travail entre les processeurs virtuels par vol de travail.

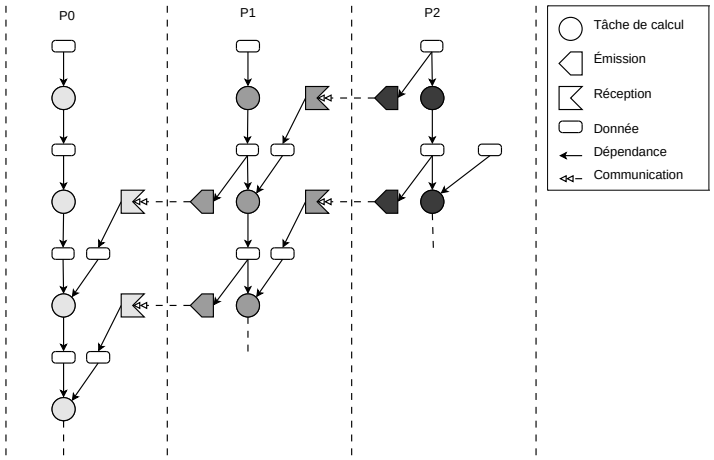

Figure 2. *Résultat de l'ordonnancement statique et distribution du graphe de flot de données sur trois processus. Chaque tâche d'émission est associée à une tâche de réception.*

3.3. *Définition d'un point de sauvegarde*

L'état d'une 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 des protocoles coordonnés est de synchroniser 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 (Chandy *et al.*, 1985).

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.

Définition 2 *Le **point de sauvegarde** de G_i est constitué du graphe de flot de données, c'est-à-dire de ses tâches et de leurs entrées associées.*

3.4. *Présentation du protocole de sauvegarde*

Le nouveau protocole de sauvegarde/reprise CCK s'inspire des méthodes de sauvegarde présentées dans (Chandy *et al.*, 1985; Koo *et al.*, 1987) : la cohérence de

l'état sauvegardé est assurée en coordonnant les processus pour vider les canaux de communication.

Comme pour le protocole standard, le protocole CCK utilise un processus coordinateur P_0 qui est extérieur au calcul. Le protocole CCK se différencie du protocole standard car il ne va envoyer un message pour vider les canaux de communication qu'entre les processus qui communiquent. La différence avec (Koo *et al.*, 1987) est que cette information indiquant si un processus va communiquer avec un autre est tirée du graphe de flot de données.

3.4.1. *Protocole sur le processus coordinateur*

La figure 3 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 (attente passive) que tous les processus P_i aient effectué leur sauvegarde pour marquer le point de sauvegarde courant comme valide.

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 :
Si le nombre de message ACK reçu est N, alors réveiller le coordinateur

---

Figure 3. *Code du coordinateur du protocole CCK*

Les processus applicatifs vont recevoir les messages envoyés et activer les différentes étapes du protocole.

3.4.2. *Protocole sur les processus applicatifs*

L'algorithme de sauvegarde pour les processus de calcul est présenté à la figure 4.

Cet algorithme comporte cinq étapes :

1. L'arrêt des calculs : il s'agit d'arrêter tous les processeurs virtuels locaux. 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 pro-

---

Sur réception d'un message M par le processus P_i :

Cas M = INIT :

1. Arrêt des calculs
   1.1 Arrêt des processeurs virtuels
   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 sur support stable
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 PING
5. Redémarrage des calculs
   5.1 Redémarrage et envoi d'un message ACK au processus coordinateur.

Cas M = PING :

Si les calculs sont arrêtés, renvoyer PONG à l'émetteur
Sinon mettre le message en attente jusqu'à l'exécution de l'étape 1.2

Cas M = PONG :

Si le nombre de messages PONG reçu est $Card(\mathcal{V})$, alors réveiller le processus

Cas M = CONT :

Si on a reçu autant de CONT que de PING, alors réveiller le processus

---

Figure 4. *Code des processus de calcul du protocole CCK*

tocole CCK qui permet de réduire son coût en nombre de messages échangés entre processus.

Le processus i analyse son graphe de flot de données G_i qui contient les futures tâches d'émission et de réception. S'il existe une tâche de réception d'une donnée provenant de P_j dans le graphe, alors un message est potentiellement en transit sur le canal de communication avec P_j.

Pour cela, le vidage des canaux de communication est réalisé 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 processeurs virtuels alors il envoie en retour le message PONG à P_i. S'il n'a pas arrêté ses processeurs virtuels, P_j enregistre le message et le message PONG ne sera envoyé à P_i qu'après l'arrêt des processeurs virtuels. Ceci permet de garantir qu'aucun message lié au calcul ne sera envoyé après le message PONG : le processus est suspendu.

3. La sauvegarde du graphe de flot de données : les processeurs virtuels 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).

5. Redémarrage des calculs : on reprend le calcul où il a été arrêté. Grâce aux messages CONT, on sait que les processus sont prêts à recevoir les futurs messages. On envoie également un message ACK au processus coordinateur pour l'informer de la fin de la sauvegarde.

3.5. *Redémarrage d'un processus défaillant*

On désire redémarrer l'application après la panne de x processus. Juste après la panne, l'application comporte deux types de processus : des processus défaillants et des processus non défaillants.

Quel que soit le processus p, sa dernière sauvegarde G_p est stockée sur un support stable. L'ensemble des sauvegardes de tous les processus constitue un état global cohérent de l'application. De plus, notons que l'état des calculs en cours sur les processus non défaillants est disponible.

En cas de panne d'un processus, le protocole standard redémarre 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 si bien qu'on demande aux processus non défaillants de ré-exécuter un ensemble de tâches de leur dernière sauvegarde afin de se ramener à un état global cohérent. Pour déterminer l'ensemble des tâches à ré-exécuter, il faut d'abord identifier l'ensemble des communications qui ont été perdues. L'ensemble des tâches à ré-exécuter est alors l'ensemble des tâches nécessaires pour émettre ces communications. Cette technique nous permet de réduire le nombre de tâches à ré-exécuter en permettant aux processus non défaillants de conserver le bénéfice des calculs déjà effectués.

Dans la suite de cette section, nous utilisons la notation X^p pour faire référence à un graphe ou à un ensemble associé au processus p, tandis que X fait référence à un

graphe ou à un ensemble global à tous les processus. X correspond à la réunion des X^p de tous les processus p.

Pour déterminer l'ensemble des tâches à ré-exécuter, il faut d'abord identifier l'ensemble des communications qui ont été perdues. L'ensemble des tâches à ré-exécuter est alors l'ensemble des tâches nécessaires pour émettre ces communications.

On propose ici un algorithme qui permet de calculer cet ensemble de tâches à ré-exécuter de manière distribuée. L'algorithme s'exécute sur chaque processus, défaillant ou non, lors d'un redémarrage. Il travaille sur le graphe de flot de données de la dernière sauvegarde, noté G^p. Pour les processus non défaillants, les informations concernant l'état d'exécution courant sont ajoutées à ce graphe. Les sommets tâches de ce graphe G^p sont donc partitionnés en deux ensembles : l'ensemble des tâches exécutées et l'ensemble des tâches non exécutées.

Une difficulté de cet algorithme distribué réside dans le fait que rejouer une communication perdue peut entraîner également la nécessité de rejouer d'autres communications, en particulier sur des processus qui ne communiquent pas directement avec les processus défaillants.

Le figure 5 montre un exemple de l'état d'une application au début de la reprise avec un processus défaillant et deux processus non défaillants. Les tâches qui ont déjà été exécutées sont grisée ; le processus défaillant, à gauche, a bien entendu perdu toutes les tâches qu'il avait exécuté.

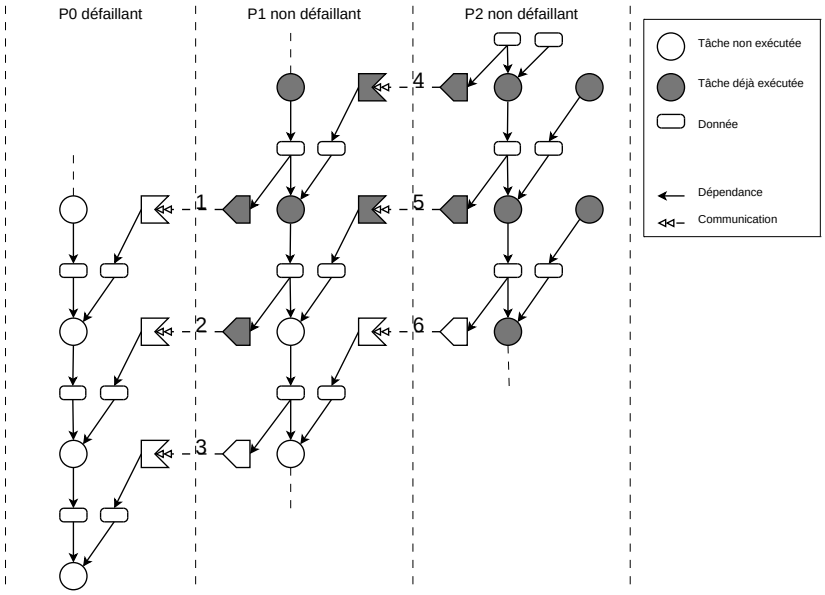

Figure 5. *Exemple avec un processus défaillant et deux processus non défaillants ; les tâches déjà exécutées sont grisées*

Les sections suivantes définisent les graphes et les ensembles qui sont utilisés dans l'algorithme présenté en section 3.5.5.

3.5.1. *Ensemble des communications perdues* $\mathcal{C}_{perdues}$

On cherche à déterminer les communications nécessaires pour rétablir un état cohérent. Plusieurs cas de figure peuvent se présenter selon l'état des processus qui communiquent (Besseron, 2006). Un seul nécessite la ré-exécution d'une tâche d'émission :

Définition 3 *L'ensemble des* ***communications perdues*** $\mathcal{C}_{perdues}$ *est l'ensemble des tâches d'émission d'un processus* p *non défaillant vers un processus défaillant qui ont déjà été exécutées depuis la dernière sauvegarde.*

Sur la figure 6, l'ensemble des communications perdues de l'exemple précédent est présenté ; ce sont les communications notées 1 et 2 sur la figure.

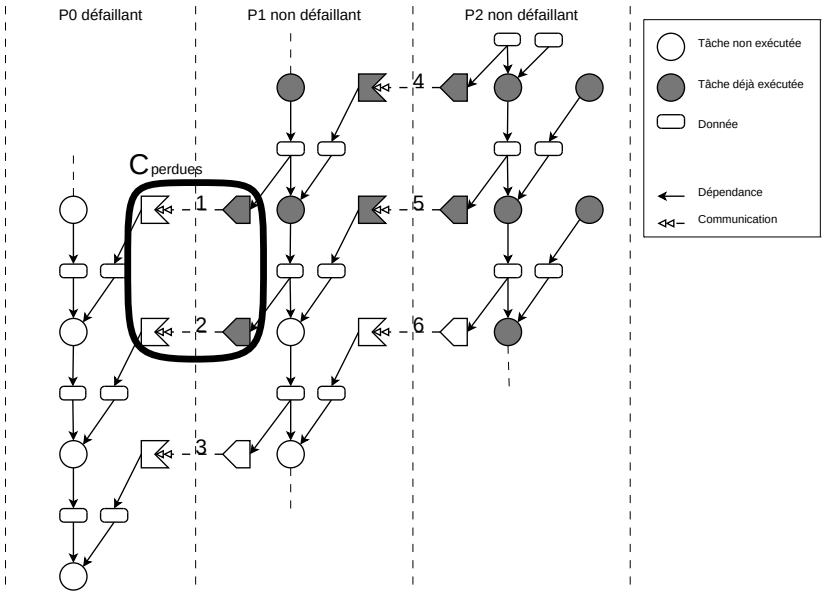

Figure 6. *L'ensemble des communications perdues*

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

Le graphe restreint aux communications $\overline{G^p}$ représente les dépendances entre les émissions et les réceptions de données au sein d'un processus p, c'est-à-dire quelles tâches de réception sont nécessaires pour ré-exécuter une tâche d'émission. Ce graphe permet de calculer l'ensemble des données à recevoir de manière à rejouer toutes

les tâches nécessaires pour ré-émettre les communications perdues vers le processus défaillant.

Définition 4 *Le **graphe restreint aux communications** $\overline{G^p}$ est le sous-graphe induit par les sommets des tâches de communication de la fermeture transitive du graphe de flot de données local G^p du processus p.*

Le graphe restreint aux communications global $\overline{G}$ est la réunion des graphes restreints locaux : $\overline{G} = \bigcup_p \overline{G^p}$.

Le graphe global $\overline{G}$ représente les précédences des communications entre tous les processus en vu de rejouer toutes les communications.

3.5.3. *Ensemble des communications à rejouer $\mathcal{C}^p_{total}$*

L'ensemble $\mathcal{C}_{perdues}$ des communications perdues contient les tâches d'émission à ré-exécuter pour rétablir l'état cohérent. Cependant, pour ré-exécuter ces tâches, il est nécessaire de ré-exécuter les tâches de calcul qui permettent de produire les données communiquées. Ces tâches de calcul peuvent nécessiter la réception de données. Le graphe global restreint aux communications $\overline{G} = \bigcup_p \overline{G^p}$ permet, grâce aux relations de précédence qu'il contient entre les tâches de communication (émission et réception), de déterminer l'ensemble total $\mathcal{C}_{total}$ des tâches d'émission à ré-exécuter pour ré-émettre toutes les communications perdues.

Définition 5 *L'ensemble des **communications à rejouer** $\mathcal{C}_{total}$ est l'ensemble des tâches de communication (émission et réception) du graphe $\overline{G}$ qui précèdent les tâches appartenant à $\mathcal{C}_{perdues}$.*

On définit alors $\mathcal{C}^p_{total}$ comme l'ensemble des tâches de $\mathcal{C}_{total}$ appartenant au processus p.

La figure 7 représente l'ensemble total des communications à rejouer ; ce sont les communications notées 1, 2 et 4 sur la figure. La communication 4 doit être rejouée car la communication 2 dépend de 4.

3.5.4. *Ensemble des tâches à ré-exécuter $\mathcal{T}^p_{\grave{a}\ ré-exécuter}$*

Une fois que chaque processus connaît son ensemble $\mathcal{C}^p_{total}$, il peut déterminer l'ensemble $\mathcal{T}^p_{\grave{a}\ ré-exécuter}$ des tâches de calcul à ré-exécuter. Ce sont les tâches nécessaires pour ré-exécuter les tâches d'émission de $\mathcal{C}^p_{total}$.

Définition 6 *L'ensemble des **tâches à ré-exécuter** $\mathcal{T}^p_{\grave{a}\ ré-exécuter}$ est l'ensemble des tâches du graphe G^p qui précèdent les tâches appartenant à $\mathcal{C}^p_{total}$.*

L'ensemble $\mathcal{T}^p_{\grave{a}\ ré-exécuter}$ contient l'intégralité des tâches à ré-exécuter pour le processus p.

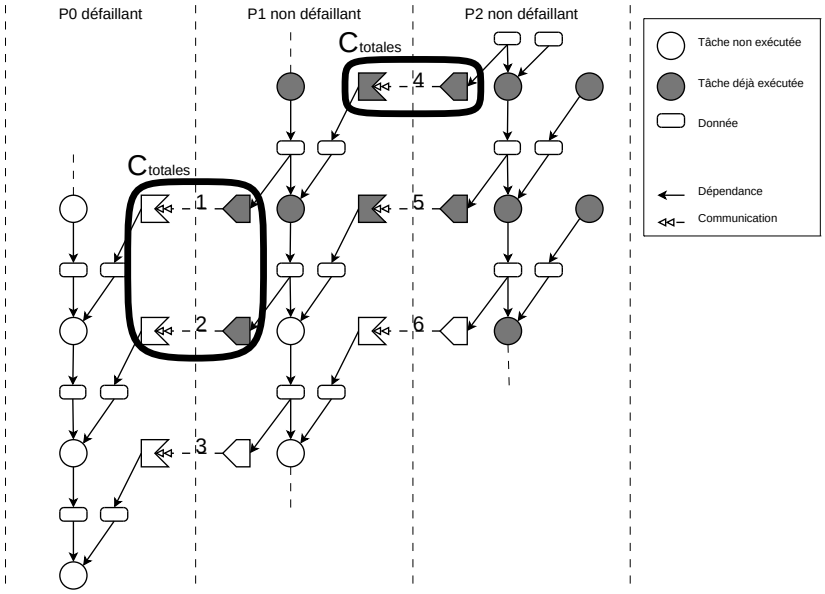

Figure 7. *L'ensemble total des communications à rejouer*

Sur la figure 8, on peut voir l'ensemble des tâches à ré-exécuter pour reprendre le calcul. Notons que $G_{défaillant}$ (le graphe du processus défaillant) doit également être ré-exécuté en totalité.

3.5.5. *Algorithme*

D'après ce qui précède, l'algorithme de reprise est le suivant et chacune de ces étapes s'effectue sur chacun des processus p de l'application :

1) Arrêt des processeurs virtuels et vidage des canaux de communication.
2) Construction du graphe restreint aux communications $\overline{G^p}$
3) Diffusion des $\overline{G^p}$ 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 $\mathcal{T}^p_{à\ ré-exécuter}$ des tâches à ré-exécuter
7) Redémarrage des processeurs virtuels

3.5.6. *Bilan*

L'ensemble des tâches à ré-exécuter $\mathcal{T}^p_{à\ ré-exécuter}$ constitue pour chaque processus p l'ensemble des calculs strictement nécessaires pour garantir la cohérence de l'état global au moment de la reprise.

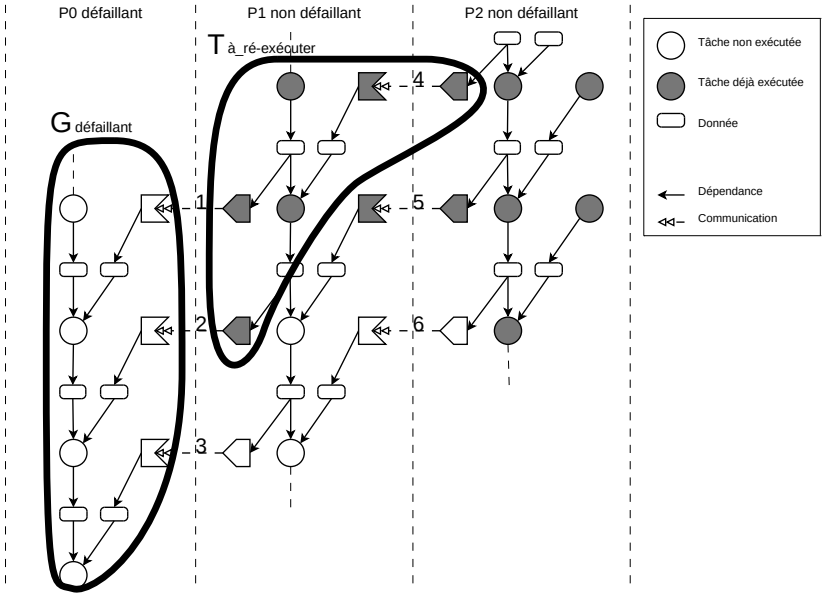

Figure 8. *L'ensemble des tâches à ré-exécuter*

Définition 7 *L'**état global reconstruit** est défini comme l'état de l'application à la fin de la phase de reprise, c'est-à-dire au moment où les processeurs virtuels redémarrent.*

L'état global reconstruit est constitué de l'union des éléments suivants :

– L'état des processus non défaillants en cours d'exécution au début de la reprise, noté $G_{exécution}$ (l'état des processus défaillants est nul).

– L'état des processus défaillants restaurés à partir de la dernière sauvegarde, noté $G_{défaillant}$.

– L'ensemble $\mathcal{T}_{à\ ré-exécuter}$ des tâches à ré-exécuter.

Rappelons qu'au moment de la reprise, les canaux de communication 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 1 *L'état global reconstruit est un état global cohérent.*

Cette proposition est prouvée dans (Besseron, 2006).

3.6. *Complexité du protocole et comparaison avec le protocole standard*

3.6.1. *Étape de sauvegarde*

Définissons tout d'abord la notion de « voisin » :

Définition 8 *Soit un processus P_i, on appelle* **voisin** *de P_i un processus P_k susceptible d'envoyer une donnée au processus P_i.*

Un processus P_i connaît ses voisins au travers des tâches de réception présentes dans son graphe de flot données. Rappelons que le protocole CCK se différencie du protocole standard (Chandy *et al.*, 1985) en envoyant moins de messages pour vider les canaux de communication (chaque processus ne se synchronise qu'avec ses voisins).

Soit N le nombre de processus et V le nombre moyen de voisins par processus, l'étape de sauvegarde des protocoles CCK et standard ont un coût en $O(NV)$ et en $O(N^2)$ respectivement. La complexité exposée par CCK est intéressante 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.

Soit

– $\mathcal{T}_N$ 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 CCK ;

– k le nombre d'étapes de sauvegarde.

Par définition, $\mathcal{T}_N^{CCK} = \mathcal{T}_N + k\mathcal{T}_{sauvegarde}^{CCK}$. Ainsi, la période de sauvegarde τ^{CCK} est égale à $\tau^{CCK} = \mathcal{T}_N^{CCK}/k$. Si on souhaite que le surcoût engendré par l'utilisation du protocole soit faible, il faut que $\mathcal{T}_N^{CCK} \approx \mathcal{T}_N$ ou que $k\mathcal{T}_{sauvegarde}^{CCK} \ll \mathcal{T}_N$. Afin que le temps de sauvegarde soit négligeable, il est nécessaire de choisir une période de sauvegarde τ^{CCK} de telle sorte que

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

3.6.2. *Étape de reprise*

Le calcul de l'ensemble des tâches à ré-exécuter se ramène essentiellement au calcul des fermetures transitives sur les graphes G^p. Le coût de cet algorithme est linéaire en fonction de la taille du graphe considéré. L'étape de vidage des canaux de communication se fait en $O(NV)$ messages et la diffusion des graphes restreints est de type *Gather-to-All* et peut se faire en $O(log_2 N)$ messages (Pigeon, 2003; Vadhiyar *et al.*, 2000).

Soit $N_{défaillants}$ le nombre de processus défaillants.

– Les processus défaillants redémarrent à partir de leur dernière sauvegarde. Le 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 des processus défaillants, on aura $\mathcal{W}^{p}_{reprise} \leq \varepsilon\tau^{cck}$ (avec $\varepsilon \leq 1$ qui représente le pourcentage de tâches à ré-exécuter sur le processus p) ; sinon $\mathcal{W}^{p}_{reprise} \approx 0$.

Le coût en travail de l'étape de reprise est donc

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

Le travail nécessaire pour la reprise est illustré sur la figure 9 en bas. Le schéma du haut montre le travail à ré-exécuter dans le cas d'un protocole standard où tous les processus redémarrent à partir de leur dernière sauvegarde.

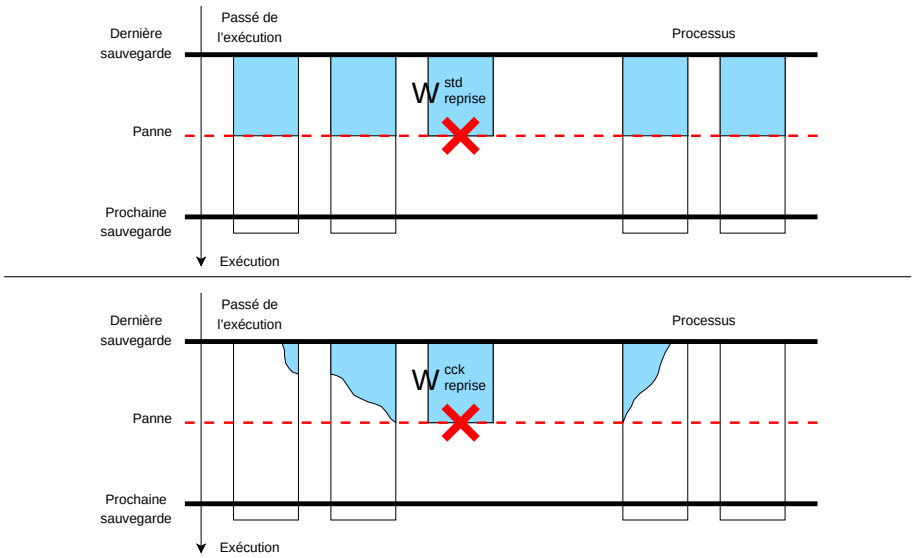

Figure 9. *Le schéma du haut montre l'ensemble du travail à ré-exécuter à la reprise pour un protocole standard de reprise globale. Celui du bas montre ce travail dans le cas du protocole CCK*

3.6.3. *Bilan*

Pour beaucoup d'applications, la complexité de notre protocole est moindre que le protocole standard sur deux points :

1) Le nombre de processus devant redémarrer suite à la défaillance d'un ou plusieurs processus est généralement moindre que le nombre total de processus. Ce

nombre peut être considéré comme une constante pour beaucoup d'applications de la classe considérée.

2) Le nombre de tâches contenues dans le *calcul strictement nécessaire* est généralement moindre que le nombre de tâches exécutées et donc à ré-exécuter.

Le premier point s'explique par la connaissance du graphe de flot de données qui nous permet de connaître les communications entre processus. Le second point est lié à la nature des dépendances de données entre les processus. Typiquement, les applications de simulation numérique qui passent à l'échelle, telle que NAMD utilisée dans (Zheng *et al.*, 2004), exposent une très bonne localité des accès aux données. Dans notre modèle, ceci implique qu'un processus a un petit nombre de voisins et que les dépendances au sein d'un processus sont faibles.

Terminons cette analyse en précisant que, grâce à notre modèle, le travail perdu qu'il est nécessaire de ré-exécuter est un sous-graphe de flot de données qui peut être redistribué sur l'ensemble des processeurs pour peu qu'il contienne suffisamment de parallélisme.

3.7. *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 pourraient s'appliquer au protocole CCK tel qu'il a été présenté.

3.7.1. *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 des données présente en mémoire 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.

3.7.2. *Ré-ordonnancement 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 ré-ordonnancement 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 ré-ordonnancement. Le résultat des simulations est présenté à la figure 12 de la section suivante.

4. Expérimentations

Dans cette section, nous présentons les mesures préliminaires obtenues avec notre prototype. Pour réaliser nos expériences, nous avons utilisé une application parallèle itérative d'une méthode de résolution de Jacobi appliquée à une surface à deux dimensions. Cette application caractérise bien le domaine d'application qui nous intéresse : les simulations de phénomènes physiques.

Pour notre application, le domaine est décomposé en éléments de taille comparable qui sont ensuite distribués sur l'ensemble des processeurs (Revire *et al.*, 2003). Après placement (Pellegrini *et al.*, 1996; Karypis *et al.*, 1997) des sous-domaines sur les processeurs, le nombre de voisins reste faible et le nombre de données à communiquer est d'environ un ordre de grandeur plus faible que le calcul local. Cette application utilisent l'interface de programmation Athapascan au dessus du moteur exécutif KAAPI.

Les expériences présentées ont été effectuées sur la grappe d'Orsay de Grid'5000 (`http://www.grid5000.fr`). Cette grappe est composée de nœuds bi-Opteron à 2 Ghz avec 2 Go de mémoire RAM et un réseau Gigabit Ethernet.

4.1. *Coût d'une étape du protocole*

La figure 10 présente le coût de coordination et de sauvegarde de notre protocole en fonction du nombre de nœuds. Cette expérience a été réalisée avec 20 et 60 serveurs de sauvegarde. Nous mesurons le temps d'exécution du protocole (points 1 à 5 de l'algorithme de la figure 4 page 10).

Avec 60 serveurs de sauvegarde, le coût d'une étape de synchronisation est linéaire. Pour 240 nœuds, le temps de synchronisation est inférieur à 230 ms, ce qui est relativement faible. Avec 20 serveurs de sauvegarde, la courbe présente une rupture de pente à partir de 80 machines. Cette rupture est due à une congestion au niveau des serveurs de sauvegarde. Pour cette expérience, la taille moyenne des états sauvegardés est de 2 Mo.

4.2. *Impact de la période de sauvegarde*

L'expérience de la figure 11 met en avant l'influence de la fréquence des étapes de sauvegarde de notre protocole sur le temps d'exécution parallèle. Nous mesurons le

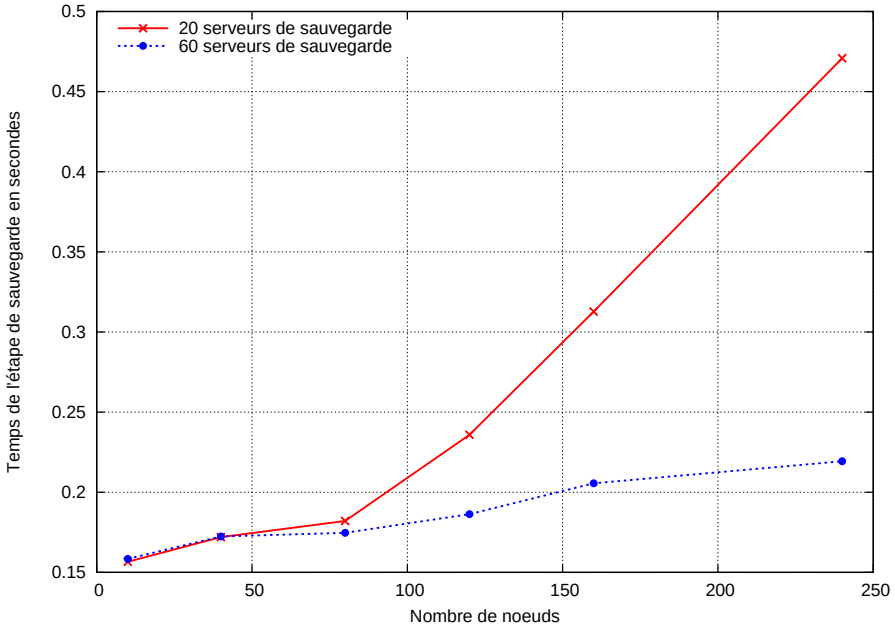

Figure 10. *Coût d'une étape de sauvegarde du protocole en fonction du nombre de processeurs avec 20 et 60 serveurs de sauvegarde*

temps d'exécution d'une instance donnée de l'application pour différentes fréquences de sauvegarde. L'expérience a été réalisée sur 32 nœuds de calculs avec 10 serveurs de sauvegarde.

Pour une fréquence inférieure à 0.1 (soit une période supérieure à 10 secondes), le surcoût est négligeable par rapport au temps d'exécution de référence (moins de 1 %). Il est bon de noter que nos travaux visent des applications de longue durée et qu'une période de sauvegarde de 10 secondes est bien plus que suffisante.

4.3. *Ré-ordonnancement local pour réduire le nombre de tâches à ré-exécuter*

La figure 12 montre l'influence de l'optimisation de ré-ordonnancement local proposée pour le protocole CCK à la section 3.7.2. L'effet de ce ré-ordonnancement dépend fortement de la date de la panne par rapport à la date du dernier point de sauvegarde.

La figure 12 nous permet de vérifier cette proposition en comparant le nombre de tâches à rejouer en fonction de la date de la panne (le point de sauvegarde intervient toutes les 0.2 UT - date normalisée -). On constate que notre optimisation augmente

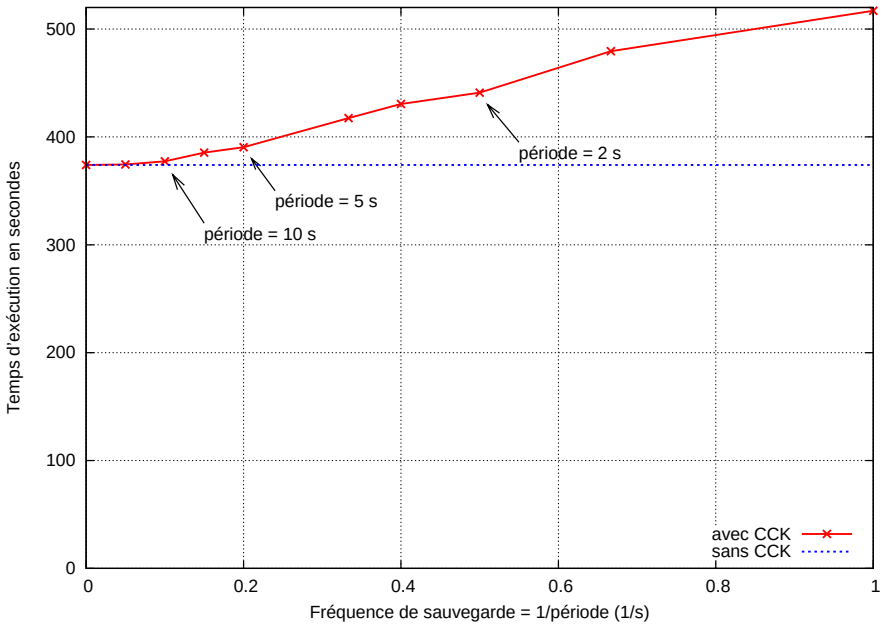

Figure 11. *Influence de la fréquence de sauvegarde sur le temps d'exécution parallèle*

largement le nombre de tâches à ré-exécuter si la panne intervient au début du calcul mais permet de ramener ce nombre à 0 pour la fin. Ceci s'explique au vu de l'objectif fixé qui est la sauvegarde des tâches engendrant les communications. Au début de l'exécution, la quasi totalité des tâches exécutées est nécessaire à la réalisation de communications. Tant que le point de sauvegarde n'est pas créé, si une panne intervient, il sera nécessaire de rejouer la quasi totalité du travail. Sans optimisation, certaines tâches exécutées n'avaient pas de dépendances directes ou indirectes avec les tâches de communication si bien qu'elles ne sont pas à rejouer quel que soit le moment où intervient la panne. Suivant cette optimisation, il existe un instant t où la quasi majorité des tâches de communication ont été enregistrées. Si une panne intervient après t, peu ou pas de tâches sont à rejouer (à l'exception de la communication en elle même). De cette manière, au fur à mesure de l'exécution, les dépendances entre les processus sont réduites et donc la quantité de travail à ré-exécuter en cas de panne diminue.

5. Conclusion et perspectives

Dans ce papier, nous avons présenté un nouveau protocole coordonné pour la sauvegarde/reprise d'applications parallèles. L'originalité de ce protocole est d'utiliser la représentation abstraite de l'exécution du programme sous la forme d'un graphe de

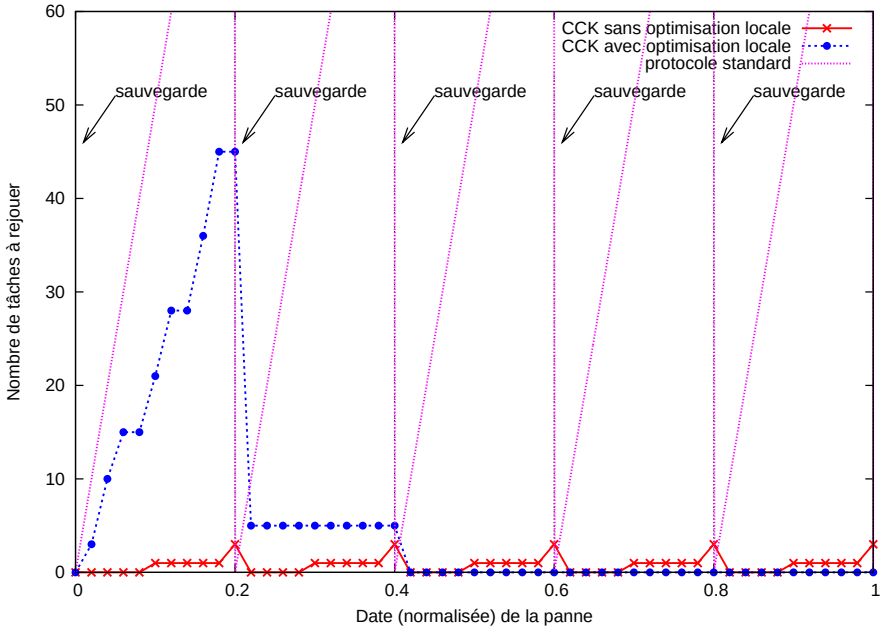

Figure 12. *Simulation du nombre de tâches à rejouer en fonction de la date de la panne avec et sans ré-ordonnancement local*

flot de données qui est distribué sur un ensemble de processus. Par comparaison avec le protocole coordonné standard, nous améliorons sa complexité sur les points suivants : 1/ durant la phase de synchronisation, moins de messages sont échangés ; 2/ le nombre de processus à redémarrer après la défaillance de l'un d'entre eux est plus faible ; 3/ le temps nécessaire pour le démarrage partiel d'un processus impliqué par la défaillance est moins important.

Ce protocole est en cours de validation dans l'environnement KAAPI. Des expériences à grande échelle sont actuellement mené sur des applications itératives ainsi que sur des applications exposant un parallélisme de données.

À terme, l'objectif est de proposer une infrastructure permettant d'adapter dynamiquement les applications parallèles itératives au nombre de ressources disponibles. Ainsi, l'ajout ou la suppression de processeurs entraînera une étape de reprise où les données seront redistribuées et le calcul restant sera repartitionné pour s'adapter au nouvel environnement, et cela sans perte importante de travail grâce au protocole CCK. Cette adaptation suit les travaux présentés dans (André *et al.*, 2004) qui laisse à la charge du programmeur la définition des points d'adaptations. Suivant notre approche, ces points d'adaptation seront déterminés dynamiquement grâce à la représentation abstraite de l'exécution qu'offre KAAPI.

6. Bibliographie

Alvisi L., Marzullo K., « Message logging : Pessimistic, optimistic, causal, and optimal », *IEEE Transactions on Software Engineering*, vol. 24, n° 2, p. 149-159, 1998.

André F., Buisson J., Pazat J.-L., « Dynamic adaptation of parallel codes : toward self-adaptable components for the Grid », *in* V. Getov, T. Kielmann (eds), *Component Models and Systems for Grid Applications*, Springer, p. 145-156, June, 2004. Proceedings of the Workshop on Component Models and Systems for Grid Applications held June 26, 2004 in Saint Malo, France. ISBN 0-387-23351-2.

Avizienis A., « Fault-Tolerant Systems. », *IEEE Trans. Computers*, vol. 25, n° 12, p. 1304-1312, 1976.

Avizienis A., Laprie J.-C., Randell B., « Dependability and its threats - A taxonomy. », *IFIP Congress Topical Sessions*, p. 91-120, 2004.

Baldoni R., « A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability », *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, IEEE Computer Society, p. 68, 1997.

Baude F., Caromel D., Delbé C., Henrio L., « A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability. », *Euro-Par*, p. 644-653, 2005.

Besseron X.*, « CCK : un protocole coordonné de sauvegarde/reprise pour la tolérance aux pannes des applications itératives en calcul numérique »*, Master's thesis, Université Joseph Fourier, Master Systèmes et Logiciels, 2006.

Bosilca G., Bouteiller A., Cappello F., Djilali S., Fédak G., Germain C., Hérault T., Lemarinier P., Lodygensky O., Magniette F., Néri V., Selikhov A., « MPICH-V : Toward a Scalable Fault Tolerant MPI for Volatile Nodes », *SuperComputing*, Baltimore, USA, 2002.

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

Chandy K. M., Lamport L., « Distributed snapshots : determining global states of distributed systems », *ACM Trans. Comput. Syst.*, vol. 3, n° 1, p. 63-75, 1985.

Elnozahy E. N. M., Alvisi L., Wang Y.-M., B. J. D., « A survey of rollback-recovery protocols in message-passing systems », *ACM Comput. Surv.*, vol. 34, n° 3, p. 375-408, 2002.

Galilée F., Roch J.-L., Cavalheiro G., Doreille M., « Athapascan-1 : On-line Building Data Flow Graph in a Parallel Language », *in* IEEE (ed.), *Pact'98*, Paris, France, p. 88-95, October, 1998.

Galilée F., Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle, Thèse de doctorat en informatique, INPG, septembre, 1999.

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

Jafar S., Gautier T., Krings A. W., Roch J.-L., « A Checkpoint/Recovery Model for Heterogeneous Dataflow Computations Using Work-Stealing. », *Euro-Par*, p. 675-684, 2005a.

Jafar S., Krings A. W., Gautier T., Roch J.-L., « Theft-Induced Checkpointing for Reconfigurable Dataflow Applications », *in* IEEE (ed.), *IEEE Electro/Information Technology Conference , (EIT 2005)*, Lincoln, Nebraska, May, 2005b. This paper received the EIT'05 Best Paper Award.

Kal L., Skeel R., Bhandarkar M., Brunner R., Gursoy A., Krawetz N., Phillips J., Shinozaki A., Varadarajan K., Schulten K., « NAMD2 : greater scalability for parallel molecular dynamics », *J. Comput. Phys.*, vol. 151, n° 1, p. 283-312, 1999.

Karypis G., Aggarwal R., Kumar V., Shekhar S., Multilevel Hypergraph Partitioning : Applications in VLSI Domain, Technical report, 1997.

Koo R., Toueg S., « Checkpointing and rollback-recovery for distributed systems », *IEEE Trans. Softw. Eng.*, vol. 13, n° 1, p. 23-31, 1987.

Pellegrini F., Roman J., Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping, Technical Report n° 1038-96, 1996.

Pigeon L.*, « Conception d'une bibliothèque pour les opérations de communication collective pour le langage de haut niveau* ATHAPASCAN *»*, Master's thesis, Dea d'Informatique : Systèmes et communications, 2003.

Pigeon L., Environnement interopérable distribué pour les simulations numériques avec composants CAPE-OPEN, PhD thesis, INPG, March, 2007.

Randell B., « System structure for software fault tolerance », *Proceedings of the international conference on Reliable software*, p. 437-449, 1975.

Revire R., Ordonnancement de graphe dynamique de tâches sur architecture de grande taille. Régulation par dégénération séquentielle et distribuée, Thèse de doctorat en informatique, INPG, septembre, 2004.

Revire R., Zara F., Gautier T., « Efficient and Easy Parallel Implementation of Large Numerical Simulation », *in* Springer (ed.), *Proceedings of ParSim03 of EuroPVM/MPI03*, Venice, Italy, p. 663-666, 2003.

Roch J. L., Gautier T., Revire R., Athapascan : API for Asynchronous Parallel Programming, Technical Report n° RT-0276, Projet APACHE, INRIA, February, 2003.

Strom R., Yemini S., « Optimistic recovery in distributed systems », *ACM Trans. Comput. Syst.*, vol. 3, n° 3, p. 204-226, 1985.

Vadhiyar S. S., Fagg G. E., Dongarra J., « Automatically Tuned Collective Communications », *Proceedings of SuperComputing2000*, November, 2000.

Wiesmann M., Pedonne F., .Schipper A., « 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)*p. 351-360, 1999.

Zara F., Faure F., Vincent J.-M., « Physical cloth simulation on a PC cluster », *in* X. P. D. Bartz, E. Reinhard (eds), *Fourth Eurographics Workshop on Parallel Graphics and Visualization 2002*, Blaubeuren, Germany, September, 2002.

Zheng G., Shi L., Kalé L. V., « FTC-Charm++ : An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI », *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September, 2004.

ANNEXE POUR LE SERVICE FABRICATION
A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :
   TSI, numéro spécial Renpar

2. AUTEURS :
   Xavier Besseron — Laurent Pigeon[2] *— Thierry Gautier — Samir Jafar*[3]

3. TITRE DE L'ARTICLE :
   Un protocole de sauvegarde / reprise
   coordonné pour les applications
   à flot de données reconfigurables

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Coordinated Checkpointing in KAAPI

5. DATE DE CETTE VERSION :
   16 décembre 2006

6. COORDONNÉES DES AUTEURS :

   – adresse postale :
   INRIA – ID (équipe MOAIS), Grenoble, France
   {Xavier.Besseron, Laurent.Pigeon, Thierry.Gautier, Samir.Jafar}@imag.fr
   – téléphone : 00 00 00 00 00
   – télécopie : 00 00 00 00 00
   – e-mail : xavier.besseron@imag.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes2.cls`,
   version 1.23 du 17/11/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél. : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com