Nouveaux Protocoles de Tolérances aux Fautes pour les Applications MPI du Calcul Haute Performance

Amina Guermouche

▶ To cite this version:

Amina Guermouche. Nouveaux Protocoles de Tolérances aux Fautes pour les Applications MPI du Calcul Haute Performance. Autre [cs.OH]. Université Paris Sud - Paris XI, 2011. Français. NNT : 2011PA112281 . tel-00666063

HAL Id: tel-00666063

https://theses.hal.science/tel-00666063

Submitted on 3 Feb 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Paris-Sud XI

Thèse

pour obtenir le grade de

Docteur de l'Université Paris-Sud XI

préparée au Laboratoire de Recherche en Informatique
dans le cadre de *l'École Doctorale d'Informatique de Paris-Sud*

présentée et soutenue publiquement

par

Amina Guermouche

6 Décembre 2011

Nouveaux protocoles de tolérance aux fautes pour les applications du calcul haute performance

---

Directeur de thèse :
Mr. Franck Cappello

Jury

Mr. André Schiper	Rapporteur
Mr. Pierre Sens	Rapporteur
Mr. George Bosilca	Examinateur
Mr. Claude Puech	Examinateur
Mr. Jean-Louis Roch	Examinateur
Mr. Marc Snir	Invité
Mr. Frédéric Vivien	Examinateur
Mr. Franck Cappello	Directeur de thèse

Table des matières

1 Introduction **7**

2 État de l'art **11**

2.1 Les Protocoles de sauvegarde de points de reprise 12
2.1.1 Les protocoles de sauvegarde de points de reprise coordonnés . . 13
2.1.2 Les protocoles de sauvegarde de points de reprise non coordonnés 14
2.1.3 Les protocoles de sauvegarde de points de reprise induits par les communications 15
2.2 Les protocoles à enregistrement de messages 15
2.2.1 Les protocoles à enregistrement de messages pessimistes 16
2.2.2 Les protocoles à enregistrement de messages optimistes 16
2.2.3 Les protocoles à enregistrement de messages causaux 17
2.2.4 Les protocoles de recouvrement arrière hiérarchiques 18

I Caractéristiques des applications du calcul haute performance **21**

3 Déterminisme des communications dans les applications du calcul haute performance **23**

3.1 Introduction 23
3.2 Standard MPI 24
3.2.1 Rappels sur quelques primitives MPI 24
3.2.2 Mise en œuvre des communications point à point dans MPI . . . 25
3.3 Modèle et définitions 26
3.3.1 Modèle 26
3.3.2 Définitions formelles 27
3.3.3 Application du modèle à MPI 27
3.4 Communications dans les applications MPI 27
3.4.1 Description des applications étudiées 27
3.4.2 Exemples de codes MPI 28
3.5 Analyse des applications 33
3.5.1 Résultats de l'analyse 34
3.6 Conclusion 35

4 Les schémas de communication dans les applications du calcul haute performance 37
4.1 Introduction 37
4.2 Étude des schémas de communications dans les applications MPI 38
4.2.1 Les études de schémas de communications existantes 38
4.2.2 Schémas de communications des primitives de communications collectives 39
4.2.3 Schémas de communications dans les applications MPI 40
4.3 Partitionnement 41
4.3.1 Les approches possibles 42
4.3.2 Partitionnement fondé sur la bissection 42
4.4 Fonctions de coût 44
4.4.1 Coût de l'enregistrement des messages α 45
4.4.2 Coût du redémarrage β 45
4.4.3 Volume des messages enregistrés L et nombre processus à redémarrer R 45
4.5 Évaluation 47
4.6 Conclusion 48

II Protocoles de tolérance aux fautes pour les applications à émissions déterministes 53

5 Un protocole de sauvegarde de points de reprise non coordonnés pour les applications à émissions déterministes 55
5.1 Introduction 56
5.2 Contexte 56
5.2.1 Modèle 56
5.2.2 Déterminisme des émissions et tolérance aux fautes 56
5.2.3 Effet domino dans les protocoles fondés sur le déterminisme des émissions 57
5.3 Description 58
5.3.1 Fonctionnement normal 58
5.3.2 Gestion des défaillances 59
5.3.3 Gestion des causalités 59
5.3.4 Pseudo code 62
5.3.5 Suppression des données obsolètes 66
5.3.6 Preuve 66
5.4 Évaluation 70
5.4.1 Description du prototype 70
5.4.2 Plate-forme d'expérimentation 72
5.4.3 Évaluation des performances en fonctionnement normal 73
5.4.4 Évaluation du nombre de processus à redémarrer : 75
5.4.5 Utilisation du protocole en définissant des groupes de processus . 76

5.5 Conclusion 80

6 HydEE : Un protocole de recouvrement arrière hiérarchique 83

6.1 Introduction 83
6.2 Description 84
6.2.1 Principes du protocole 84
6.2.2 Défaillance et redémarrage 84
6.2.3 Suppression des données obsolètes 92
6.2.4 Processus de redémarrage distribué 92
6.2.5 Preuve 92
6.3 Évaluation 95
6.3.1 Description du prototype 96
6.3.2 Configuration 96
6.3.3 Performances en exécution sans faute 97
6.3.4 Évaluation des performances en redémarrage 99
6.4 Conclusion 100

7 Conclusion 103

Chapitre 1

Introduction

Le domaine du calcul haute performance a montré une constante et rapide progression de la puissance des calculs. Si cette tendance se poursuit, des machines avec une puissance de calcul exaflopique devraient voir le jour à la fin de la décennie. Une telle puissance requiert beaucoup plus de composants (nœuds, cœurs, mémoire) que les systèmes actuels. Les projections fournies dans les rapports IESP [1] estiment ainsi que le nombre de cœurs augmentera de trois ordres de grandeur.

En augmentant le nombre de ces composants, la fréquence des défaillances augmente également. Sur de telles machines, le temps moyen entre les défaillances ($MTBF$) devrait être de quelques heures seulement. La tolérance aux fautes représente donc une des principales questions à étudier pour l'exascale et au delà.

La tolérance aux fautes automatique a pour avantage de réduire la difficulté du développement des applications en masquant les défaillances et leur gestion à l'utilisateur. Les approches les plus utilisées sont la réplication et la tolérance aux fautes par points de reprise. La réplication consiste à répliquer chaque processus de l'application sur un ensemble de machines, consommant ainsi un nombre de ressources (unités de calcul, énergie) proportionnel au degré de la réplication (la réplication nécessite en plus des contrôles supplémentaires). Compte tenu de la nécessité de limiter les coûts de réalisation des systèmes de calcul haute performance et de leur consommation, la réplication n'est pas considérée comme une solution viable pour l'exascale. La technique par points de reprise consiste à capturer l'état de chaque processus du système durant l'exécution. Lorsqu'une défaillance se produit, les processus fautifs récupèrent l'état sauvegardé et redémarrent à partir de celui-ci. Cependant, un ensemble cohérent de points de reprise doit être trouvé afin de garantir que l'exécution des processus soit correcte après redémarrage. Plusieurs techniques existent pour assurer que les processus redémarrent d'un état qui garantit la correction de l'exécution. Celles-ci sont fondées soit sur les points de reprise coordonnés soit sur les points de reprise induits par les communications ou encore sur les points de reprise non coordonnés avec ou sans enregistrement des messages.

Dans les protocoles de sauvegarde de points de reprise coordonnés, la sauvegarde des points de reprise se fait de façon coordonnée sur tous les processus. Lorsqu'une défaillance se produit, tous les processus redémarrent à partir du point de reprise le plus récent. Afin d'éviter la coordination, dans les protocoles de sauvegarde de points

de reprise non coordonnés, les processus sauvegardent leur image indépendamment les uns des autres. Cependant, après une défaillance, il existe un risque que les processus se voient obliger de revenir au tout début de l'exécution. Afin de remédier à ce problème, cette technique est associée à un protocole à enregistrement de messages. Dans un tel protocole, l'ensemble des messages échangés entre les processus ainsi que l'ordre de d'émission et de réception de tous les messages (appelés déterminants) sont enregistrés. Lorsqu'une défaillance se produit, le processus fautif redémarre du dernier point de reprise et rejoue les messages enregistrés dans l'ordre indiqué par les déterminants.

Ces protocoles ont pour inconvénients de redémarrer tous les processus ou d'enregistrer tous les messages. Sur un nombre de processus important, ces deux techniques deviennent rapidement très coûteuses. Des solutions alternatives doivent donc être trouvées. Une première direction est d'étudier les caractéristiques des plates-formes et des applications utilisées dans le but d'explorer des propriétés qui pourraient être exploitées pour éviter ou limiter le coût des protocoles de tolérance aux fautes.

Dans ce document, nous nous intéressons aux applications MPI. De récentes études ont montré que les bibliothèques MPI possèdent un certain degré de déterminisme, et qu'il est possible d'améliorer les protocoles à enregistrement de messages en n'enregistrant que les déterminants des évènements non déterministes [2].

Nous nous intéressons ici à des caractéristiques plus générales, qui se rapportent aux applications, et nous proposons de nouveaux protocoles de tolérance aux fautes qui pallient les limites des protocoles existants en se fondant sur ces caractéristiques.

Contributions

Les contributions présentées dans cette thèse se déclinent en deux parties. La première représente l'étude des applications MPI du calcul haute performance afin d'extraire les caractéristiques de celles-ci. Dans la seconde partie, nous proposons deux nouveaux protocoles de tolérance aux fautes fondés sur ces caractéristiques.

Caractéristiques des applications MPI

Les protocoles de tolérance aux fautes sont fondés sur des suppositions qui ne correspondent pas forcément aux spécificités des applications MPI. Dans cette partie, nous étudions le déterminisme et les schémas de communications de celles-ci.

Déterminisme des applications MPI Les protocoles de tolérance aux fautes supposent que les exécutions des applications sont soit non déterministes (protocoles de sauvegarde de points de reprise coordonnés, induits par les communications et non coordonnés simple), soit déterministes par morceaux (protocoles à enregistrement de messages). Dans cette thèse, nous étudions les communications dans 26 applications MPI du calcul haute performance. Nous mettons en avant une nouvelle caractéristique appelée *déterminisme des émissions* : Pour toute exécution correcte et compte tenu d'un ensemble de paramètres d'entrée donné, la séquence des émissions est toujours la même pour chacun des processus, quel que soit l'ordre de réception des messages.

Communications par groupes dans les schémas de communications MPI Dans les protocoles de tolérance aux fautes, le même protocole est utilisé par tous les processus sans exploiter les schémas de communications des processus. Plusieurs travaux ont étudié ces schémas dans les applications MPI [3, 4]. Dans ce document, nous nous fondons sur eux afin de former des groupes de processus. En effet, dans beaucoup d'applications, les processus communiquent plus avec certains processus qu'avec d'autres. Nous étudions, dans un premier temps, les primitives collectives, où tous les processus sont impliqués, afin de vérifier si les algorithmes utilisés possèdent également cette caractéristique. Par la suite, nous étudions les schémas de communications de plusieurs applications MPI.

Protocoles de tolérance aux fautes pour les applications MPI

À partir de ces deux caractéristiques, nous proposons deux protocoles de tolérance aux fautes pour les applications MPI. Les deux protocoles ont été mis en œuvre dans la bibliothèque MPICH2 et les expérimentations ont été réalisées sur la plate-forme Grid'5000.

Un protocole de sauvegarde de points de reprise non coordonnés pour les applications à émissions déterministes En garantissant que la même séquence de messages est émise d'une exécution à une autre, le déterminisme des émissions permet d'éviter la gestion des messages émis par les processus fautifs en particulier la création des messages orphelins. En se fondant sur cette caractéristique, nous proposons un protocole de sauvegarde de points de reprise non coordonnés qui pallient les problèmes des protocoles non coordonnés existants en n'enregistrant qu'un sous-ensemble des messages échangés et leur date d'émission sur la mémoire de l'émetteur. Il a pour avantage de ne pas forcer de redémarrage global et de n'enregistrer aucun déterminant.

Nous avons également adapté le protocole afin de l'utiliser sur des groupes de processus pour limiter le nombre de processus à redémarrer.

HydEE : Un protocole de tolérance aux fautes hiérarchique pour les applications à émissions déterministes En se fondant sur les groupes de processus et afin de limiter au maximum le nombre de processus effectuant un retour arrière, nous proposons HydEE, un protocole de tolérance aux fautes hiérarchique qui applique un protocole de sauvegarde de points de reprise coordonnés au sein des groupes et un protocole à enregistrement de messages entre les groupes. Les messages inter-groupe et leur date d'émission sont enregistrés sur la mémoire de l'émetteur. Grâce au déterminisme des émissions, aucun déterminant n'est sauvegardé. HydEE a pour avantage de confiner les défaillances en ne redémarrant que les processus des groupes où elles se produisent et de limiter le volume des données enregistrées.

Ces deux protocoles pallient les problèmes des protocoles de tolérance aux fautes existants :

1. seul un sous-ensemble des processus de l'application redémarrent.

2. seul un sous-ensemble des messages sont enregistrés sur la mémoire des émetteurs et aucun déterminant n'est sauvegardé.

Organisation du document

Ce document est organisé comme suit. Le Chapitre 2 est consacré à l'étude de l'état de l'art sur les protocoles de tolérance aux fautes existants. La première partie présente les caractéristiques des applications MPI. Le Chapitre 3 présente le modèle sur lequel nous nous fondons et l'étude du déterminisme des émissions. Dans le Chapitre 4, nous exposons les différents schémas de communications étudiés. Nous présentons également un outil de partitionnement et les résultats de deux regroupements différents obtenus grâce à cet outil.

Dans la seconde partie, nous présentons les deux protocoles de tolérance aux fautes. Le Chapitre 5 détaille le protocole de sauvegarde de points de reprise non coordonnés. Le Chapitre 6 présente Hydee. Dans ces deux chapitres, nous présentons l'évaluation des protocoles réalisée sur Grid'5000 après leur mise en œuvre dans la bibliothèque MPICH2.

Nous concluons par un chapitre dressant un bilan des travaux exposés et présentant les futurs travaux possibles.

Chapitre 2

État de l'art

Sommaire

2.1 Les Protocoles de sauvegarde de points de reprise 12
2.1.1 Les protocoles de sauvegarde de points de reprise coordonnés 13
2.1.2 Les protocoles de sauvegarde de points de reprise non coordonnés 14
2.1.3 Les protocoles de sauvegarde de points de reprise induits par les communications 15
2.2 Les protocoles à enregistrement de messages 15
2.2.1 Les protocoles à enregistrement de messages pessimistes . . . 16
2.2.2 Les protocoles à enregistrement de messages optimistes . . . 16
2.2.3 Les protocoles à enregistrement de messages causaux 17
2.2.4 Les protocoles de recouvrement arrière hiérarchiques 18

Selon le système, les fautes considérés et les caractéristiques des applications (déterminisme par exemple), les techniques de tolérance aux fautes diffèrent. Dans ce document, nous nous intéressons aux calculs par passage de messages. Les systèmes que nous considérons sont asynchrones. Nous ne décrivons donc pas les techniques de tolérance aux fautes pour des systèmes synchrones. De plus, nous considérons des pannes franches, dans lesquelles la défaillance d'un processus implique son arrêt total jusqu'à la fin de l'exécution. Nous n'étudierons donc pas les protocoles de tolérance aux fautes pour d'autres catégories de pannes (les pannes byzantines par exemple).

La tolérance aux fautes peut être soit imposée par l'utilisateur, soit automatique. La première classe s'utilise au niveau de l'application. En effet, la technique de tolérance aux fautes est directement introduite dans le code de l'application. Par exemple, dans [5], des mécanismes supplémentaires sont ajoutés dans le code de l'application afin de gérer les différents types d'erreurs. Des points de reprise peuvent également être utilisés au niveau applicatif. Cette technique est présente dans plusieurs applications du calcul haute performance telles que *Lammps* [6] et *MILC* [3]. L'avantage est que le programmeur peut décider de sauvegarder le point de reprise au moment où la quantité de données à sauvegarder est moins grande et de savoir quelles sont les données à enregistrer. Cependant, cette technique nécessite l'intervention du programmeur [7]. De plus, il faut avoir accès aux codes des applications qui ne sont pas toujours disponibles.

Les techniques de tolérance automatique les plus utilisées sont la réplication et le recouvrement arrière.

La réplication réplique l'exécution n fois et tolère donc au plus n défaillances. Cependant, cette technique divise par n le nombre de ressources de calcul disponible pour un nombre de ressources fixé.

Le recouvrement arrière consiste à sauvegarder l'image des processus et de les redémarrer à partir de celle-ci s'ils subissent une défaillance.

Dans la suite du document, nous étudions les protocoles par recouvrement arrière. Il existe deux familles de protocoles utilisant cette technique : les protocole de sauvegarde de points de reprise et les protocoles à enregistrement de messages.

Dans [8], les auteurs ont décrit les principes des principaux protocoles de tolérance aux fautes existants.

2.1 Les Protocoles de sauvegarde de points de reprise

Les protocoles de tolérance aux fautes fondés sur les points de reprise sont les plus utilisés. Ils consistent à sauvegarder périodiquement l'état des processus sur un support stable (support qui tolère les défaillances).

Ces protocoles supposent un modèle d'exécution non déterministe où il est impossible de garantir le rejeu à l'identique des évènements après une défaillance.

Lorsqu'une défaillance se produit, le processus fautif restaure son état à partir de la dernière image sauvegardée sur support stable.

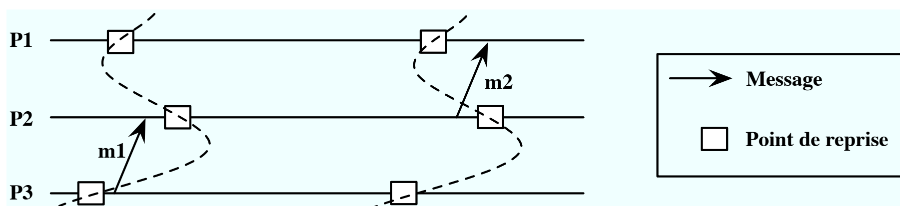


FIGURE 2.1 – État global cohérent

Cependant, certaines précautions doivent être prises pour que la réexécution soit cohérente. En effet, redémarrer un processus à partir d'un état peut créer des messages dits "orphelins".

Message orphelin : Un message est dit orphelin s'il a été reçu mais pas encore émis. Cela se produit lorsqu'après une défaillance, l'émetteur du message effectue un retour arrière alors que son récepteur ne redémarre pas ou redémarre après le réception de ce message.

Le risque d'incohérence de l'exécution après redémarrage est dû au fait que ces protocoles supposent que l'exécution des applications est non déterministe. De ce fait, après

une défaillance, il n'y a aucun moyen de garantir que les messages précédemment émis seront émis à nouveau. Sur la figure 2.1, si le processus p_3 subit une défaillance juste après l'émission du message m_1, en effectuant un retour arrière, le message m_1 est vu comme reçu par le processus p_2 mais pas émis par p_3.

L'état des processus à ce moment n'est pas un état cohérent car il n'est pas possible de croiser, dans une exécution sans fautes, une configuration telle où un message est reçu mais pas émis.

État global cohérent : L'état global correspond à l'état de tous les processus et de l'ensemble des canaux. C'est un état qui peut être rencontré durant une exécution sans fautes. D'après la définition décrite dans [9], un état global cohérent est un état tel que, si l'état d'un processus reflète la réception d'un message alors l'envoi du message est dans l'état de son émetteur. C'est-à-dire, s'il n'y a pas de message orphelin. L'ensemble des états des processus après la récupération de l'image de certains à partir d'un point de reprise est appelé "ligne de recouvrement" [10].

Sur la figure 2.1, l'état formé par les premiers points de reprise n'est pas un état cohérent (à cause du message orphelin m_1) contrairement à l'état formé par les deuxièmes points de reprise. En effet, si le processus p_2 subit une défaillance après son deuxième point de reprise, le message m_2 est vu comme envoyé par p_2 mais peut ne pas être encore reçu par p_1. Mais cette configuration est possible dans une exécution sans fautes si les canaux sont asynchrones.

Les protocoles de sauvegarde de points de reprise sont classifiés en trois famille : coordonnés, non coordonnés et induits par les communications.

2.1.1 Les protocoles de sauvegarde de points de reprise coordonnés

Dans ce type de protocoles, les points de reprise sont coordonnés sur tous les processus et sauvegardés sur support stable afin de former un état global cohérent. Lorsqu'une défaillance se produit, tous les processus effectuent un retour arrière vers le dernier point de reprise. Dans ce type de protocole, seul le dernier point de reprise est nécessaire, il n'est donc pas utile de garder les autres.

Dans [11], les auteurs ont proposé une première façon de réaliser la coordination, en bloquant les processus pendant la phase de sauvegarde du point de reprise. Lorsqu'un processus décide de sauvegarder un point de reprise, il diffuse une requête vers tous les autres processus. Lorsqu'un processus reçoit une telle requête, il arrête son exécution, prend un premier point de reprise et envoie une réponse au processus initiateur. Lorsque le processus initiateur reçoit toutes les réponses des autres, il diffuse un autre message pour sauvegarder le nouveau point de reprise et supprimer le précédent. Puis, le processus reprend son exécution. Cependant, dans [12], les auteurs ont montré que pour des applications où la taille des points de reprise à sauvegarder est grande, le surcoût introduit sur le temps d'exécution en bloquant les processus est important.

Le protocole de sauvegarde de points de reprise coordonnés le plus utilisé est celui présenté dans [9]. Dans ce protocole, lorsqu'un processus sauvegarde son état local, il

diffuse un marqueur sur tous les canaux sortants. Lorsqu'un processus reçoit un tel message pour la première fois, il prend son état local et sauvegarde l'état de l'ensemble des canaux entrants. Il diffuse également le marqueur. Ce protocole suppose que les canaux sont FIFO afin que les processus sachent à quel moment ils doivent prendre l'état des canaux. Dans le cas où les canaux ne sont pas FIFO, les marqueurs sont attachés aux messages émis après la prise de l'état [13].

Afin de limiter le nombre de processus impliqués dans la coordination, les auteurs de [14] ont proposé un protocole qui ne coordonne que les processus ayant communiqué depuis le dernier point de reprise. De plus, lorsqu'une défaillance se produit, seuls les processus ayant communiqué effectuent un retour arrière.

D'autres protocoles sont fondés sur le temps pour coordonner les processus. Les protocoles proposés dans [15, 16, 17, 18] supposent que les horloges sont approximativement synchronisées. Chaque processus sauvegarde son état lorsque l'horloge locale atteint le moment de sauvegarde du point de reprise.

2.1.2 Les protocoles de sauvegarde de points de reprise non coordonnés

Dans les protocoles de sauvegarde de points de reprise non coordonnées, les processus sauvegardent leur état indépendamment les uns des autres. Lorsqu'une défaillance se produit, le processus défaillant redémarre à partir de son dernier point de reprise. Cependant, afin de recevoir les messages orphelins, des processus peuvent avoir à revenir au point de reprise précédant leur réception.

Plusieurs techniques de calcul de la ligne de recouvrement existent. Elles sont fondées sur les graphes de dépendance entre les points de reprise [19, 20].

Sur la figure 2.2, si le processus p_2 subit une défaillance après l'émission du message m_4, le processus p_3 doit redémarrer de son dernier point de reprise à cause du message orphelin m_4.

En forçant le retour arrière des processus ayant reçu un message orphelin, les processus peuvent être amenés à faire des retours arrière en cascade jusqu'à atteindre le début de l'exécution. Un tel phénomène est appelé “effet domino” [10]. Sur la figure 2.2, si le processus p_1 subit une défaillance après l'émission du message m_3, le processus p_2 doit revenir avant son dernier point de reprise, et donc p_3 va également effectuer un retour arrière. Puis p_1 doit faire un nouveau retour arrière pour recevoir le message orphelin m_2 ce qui conduit p_3 à faire de même pour recevoir le message m_1.

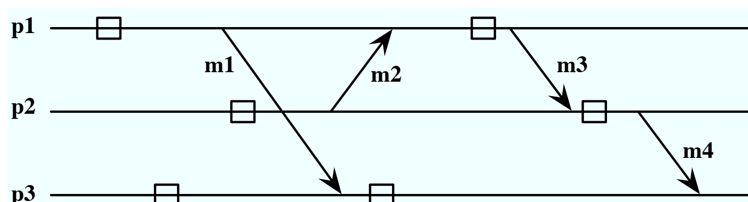


FIGURE 2.2 – Effet domino

Ces protocoles ont également pour inconvénient de compliquer la suppression des données obsolètes car le dernier point de reprise peut ne pas être suffisant à la construction de la ligne de recouvrement.

2.1.3 Les protocoles de sauvegarde de points de reprise induits par les communications

Dans ce type de protocole, les processus sauvegardent leur état local indépendamment les uns des autres. Des données supplémentaires sont attachées aux messages afin de forcer la prise de points de reprise et d'éviter donc l'effet domino ([21, 22, 23]). Cependant, des études ont montré que le nombre de points de reprise forcés était plus élevé que le nombre de prises d'état local [24]. Ces protocoles ne sont donc pas adaptés à l'exécution d'applications du calcul haute performance puisque la maîtrise du nombre de points de reprise est un élément important pour limiter le surcoût de la tolérance aux fautes.

2.2 Les protocoles à enregistrement de messages

Dans ce type de protocoles, le contenu de chaque message est enregistré. Ces protocoles supposent également un modèle d'exécution non déterministe, mais contrairement aux protocole de sauvegarde de point de reprise, ils supposent qu'il est possible d'identifier les évènements non déterministes et de les enregistrer afin de les rejouer à l'identique après une défaillance. De telles applications sont dites déterministe par morceaux. L'exécution est donc constituée d'un ensemble d'intervalles d'évènements déterministes commençant par des évènements non déterministes (les réceptions) [25]. Ainsi, tant que les messages sont reçus dans le même ordre, l'exécution est la même. Pour garantir l'ordre des réceptions, chaque message est identifié par un quadruplet, appelé "déterminant", contenant l'identité de l'émetteur, un identifiant unique du message attribué par l'émetteur (qui peut être le numéro de séquence de son émission), l'identité du récepteur et un identifiant unique attribué par le récepteur (qui peut être son numéro de séquence de réception) [26].

Lorsqu'une défaillance se produit, le processus fautif effectue un retour arrière. Les messages enregistrés lui sont renvoyés par les processus sans que ceux-ci n'effectuent de retour arrière et sont reçus dans l'ordre indiqué par les déterminants. Afin d'éviter, d'une part un retour arrière vers le début de l'exécution et d'autre part de garder en mémoire tous les messages échangés pendant l'exécution, ces protocoles sont associés à un protocole de sauvegarde de points de reprise non coordonnés.

L'enregistrement des messages se fait soit par l'émetteur soit par le récepteur. Dans les protocoles fondés sur le récepteur, les processus sauvegardent chaque message qu'ils reçoivent sur support stable [27].

Dans les protocoles à enregistrement de messages fondés sur l'émetteur, chaque processus enregistre les messages dans sa mémoire. Ces données sont enregistrées sur support stable lorsque le processus sauvegarde son point de reprise [28]. L'évaluation

présentée dans [29] montre que les performances en exécutions sans fautes des protocoles fondés sur l'émetteur sont meilleures que celles fondées sur le récepteur.

Il existe trois familles de protocoles à enregistrement de messages : optimistes, pessimistes et causaux. Ils se différencient par leur gestion des déterminants.

2.2.1 Les protocoles à enregistrement de messages pessimistes

Les protocoles pessimistes supposent qu'une défaillance peut se produire immédiatement après chaque réception. Dans ce type de protocoles [28, 29], tous les déterminants sont sauvegardés sur support stable de façon synchrone : un processus ne peut envoyer un message que si les déterminants des messages dont ce message dépend sont stockés sur support stable. Cela garantit que les messages orphelins sont toujours réémis étant donné que tous les messages peuvent être reçus dans le bon ordre car leurs déterminants sont sur support stable.

Lorsqu'une défaillance se produit, seul le processus fautif effectue un retour arrière vers son dernier point de reprise.

Ces protocoles ont plusieurs avantages. D'une part, seuls les processus fautifs redémarrent après une défaillance. De plus, ils permettent un recouvrement facile puisqu'aucun message orphelin n'est créé. D'autre part, étant donné qu'un processus fautif effectue un retour arrière vers son dernier point de reprise, il n'est pas nécessaire de garder les messages qui ont été reçus avant ce point de reprise, facilitant ainsi la suppression des données obsolètes.

Cependant, ces protocoles introduisent un surcoût élevé dans des applications où les processus communiquent beaucoup. Ceci est dû à l'enregistrement synchrone des déterminants [30].

2.2.2 Les protocoles à enregistrement de messages optimistes

Les protocoles optimistes [25, 27, 31, 32, 33, 34] tentent d'améliorer les performances en exécution sans fautes des protocoles à enregistrement de messages en enregistrant les déterminants sur support stable de façon asynchrone : les déterminants sont gardés en mémoire et sont périodiquement enregistrés sur support stable.

Lorsqu'une défaillance se produit, le processus fautif effectue un retour arrière vers son dernier point de reprise, perdant ainsi l'ensemble des déterminants qui se trouvaient dans sa mémoire. Ceci peut donc amener à la création de messages orphelins. Les processus ayant reçu de tels messages effectuent également un retour arrière. Sur la figure 2.3, si le processus p_2 subit une défaillance, il effectue un retour arrière vers son dernier point de reprise. Étant donné que les déterminants des messages m_1 et m_2 sont sauvegardés dans sa mémoire volatile, ils sont perdus après son redémarrage. Il ne peut plus ordonner la réception des deux messages et le messages orphelin m_3 peut ne pas être réémis, ce qui force le redémarrage du processus p_1.

Cependant, dans [31], les auteurs ont montré que ces protocoles ne provoquent pas d'effet domino.

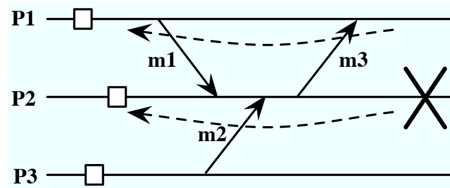


FIGURE 2.3 – Redémarrage avec un protocole optimiste

Afin de détecter les processus à redémarrer, les protocoles à enregistrement de messages optimistes tracent le chemin des dépendances causales entre les messages. Pour ce faire, les processus sauvegardent des vecteurs de dépendances [25, 27, 27, 35, 36] ou des vecteurs d'horloges tolérantes aux fautes [32]. Ces vecteurs contiennent autant d'entrées qu'il y a de processus et ils sont attachés à chaque message émis.

Ces protocoles ont également pour inconvénients de compliquer la suppression des données obsolètes car plusieurs processus peuvent être forcés à redémarrer d'un point de reprise qui n'est pas le plus récent après la défaillance d'un seul processus.

2.2.3 Les protocoles à enregistrement de messages causaux

Ces protocoles [26, 37, 38, 39] combinent les avantages de ceux décrits précédemment : les déterminants sont enregistrés de manière asynchrone en mémoire volatile et tous les messages orphelins sont réémis. Pour ce faire, chaque processus maintient une table contenant les dépendances causales de ses messages [40]. Lorsqu'il émet un message, il y attache les déterminants dont ce message dépend. Cela garantit que tous les processus dépendant d'un évènement non déterministe possèdent les déterminants nécessaires pour éviter les messages orphelins.

Lorsqu'une défaillance se produit, seul le processus fautif effectue un retour arrière et rejoue les messages dans l'ordre grâce aux déterminants sauvegardés soit sur support stable, soit au niveau des autres processus. Sur la figure 2.4, lorsque le processus p_2 subit une défaillance au niveau de la marque, il effectue un retour arrière vers son dernier point de reprise. Le déterminant du message m_4 est disponible au niveau du processus p_1 car il est attaché au message m_5.

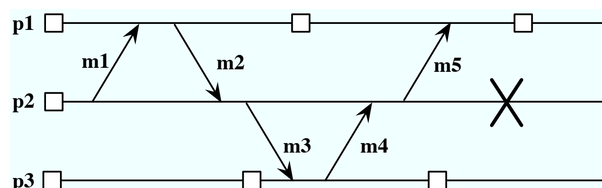


FIGURE 2.4 – Redémarrage avec protocole causal

Si tous les processus dépendant d'un même évènement non déterministe subissent

une défaillance, tous les déterminants dont ils ont besoin sont perdus. Cependant, étant donné qu'ils redémarrent tous, aucun message orphelin n'est créé et ainsi l'état qu'ils atteindront sera cohérent.

Dans ce type de protocole, un processus fautif p n'effectue pas de retour arrière vers un point de reprise autre que son dernier. Il n'est donc pas nécessaire, pour un processus q qui a envoyé un message à p et qui est reçu avant ce point de reprise, de garder ce message ni de transmettre les causalités pour celui-ci. Ainsi donc, les processus s'échangent périodiquement l'index du point de reprise courant [40].

Dans [41], les auteurs ont montré que le surcoût introduit par ce protocole est principalement dû aux dépendances causales attachées aux messages et à leur gestion. Ainsi, plus la taille de l'exécution est grande en nombre de processus, plus la taille des données attachées aux messages augmente.

Comme dit précédemment, les protocoles optimistes et causaux attachent des données supplémentaires sur les messages introduisant un surcoût réduisant les performances. Afin de réduire la taille des données attachées, il est possible de sauvegarder les déterminants sur support stable en utilisant une interface appelée "enregistreur d'évènements" [30]. Ce concept améliore les performances des protocoles causaux [41] et optimistes [42]. Cependant ces performances sont limitées à grande échelle [42]. Pour les améliorer, les auteurs de [43] ont proposé un enregistreur d'évènements distribué. Mais, pour des applications où le nombre de messages est élevé, le surcoût introduit par l'enregistreur distribué est important [43].

Afin de réduire le surcoût introduit pas l'enregistrement des déterminants, dans [2], les auteurs ont proposé de raffiner le modèle d'exécution déterministe par morceaux pour prendre en compte la sémantique de l'interface MPI. Dans les travaux considérant une exécution déterministe par morceaux précédemment cités, un évènement non déterministe est associé à la réception de chaque message. Cependant, la plupart des primitives de communications MPI ont un comportement déterministe. Seul l'émission d'une requête de réception de message dans laquelle l'émetteur du message n'est pas précisée (utilisation de *MPI_ANY_SOURCE*), ou la complétion asynchrone d'une requête de réception (par exemple, utilisation de *MPI_Wait_Any()*) implique un comportement non déterministe. Ainsi, en utilisant ce modèle, les protocoles à enregistrement de messages ont beaucoup moins d'évènements non déterministes à enregistrer que dans le modèle classique, ce qui permet d'améliorer leurs performances [44].

2.2.4 Les protocoles de recouvrement arrière hiérarchiques

Dans les protocoles décrits précédemment, tous les processus appliquent la même technique de tolérance aux fautes. Plusieurs travaux proposent de combiner différents protocoles de recouvrement arrière dans un protocole hiérarchique afin de remédier aux problèmes des protocoles existants. Des groupes de processus sont créés. Un premier protocole est utilisé entre les groupes de la même manière que les protocoles appliqués aux processus, et un second protocole est utilisé au sein des groupes.

Certains de ces protocoles sont fondés sur l'aspect hiérarchique des architectures visées où un groupe peut être représenté par une grappe [45, 46], d'autres sur les propriétés des schémas de communications des applications où les processus qui communiquent le plus sont placés dans un même groupe [47, 48] ou encore sur la probabilité de défaillance des processus où les processus qui ont la même probabilité de défaillir sont dans un même groupe (typiquement des processus s'exécutant sur les cœurs d'un même nœud) [49]. Ces protocoles se différencient également par le protocole utilisé entre les groupes. En effet, ils utilisent tous un protocole de sauvegarde de points de reprise coordonnés au sein des groupes afin de faciliter le redémarrage et d'améliorer les performances en exécution sans fautes. Un état global cohérent est assuré soit en utilisant un protocole de sauvegarde de points de reprise entre les groupes soit en enregistrant les messages inter-groupe.

2.2.4.1 Les protocoles de sauvegarde de points de reprise inter-groupe

Dans ces protocoles, l'état global cohérent est assuré en utilisant un protocole de sauvegarde de points de reprise entre les groupes.

Dans [45] et [46], les auteurs considèrent des grilles de types fédérations de grappes. Les sauvegardes de points de reprise entre les groupes sont faites en utilisant un protocole de sauvegarde de points de reprise induits par les communications. Le but de ce protocole est de profiter du fait qu'il y a plus de communications au sein des grappes qu'entre elles, diminuant ainsi le nombre de points de reprise forcés comme c'est le cas dans les protocoles de sauvegarde de points de reprise induits par les communications. Des informations supplémentaires sont ajoutées aux messages inter-groupe afin de forcer la sauvegarde du point de reprise si nécessaire. Afin d'éviter de forcer le retour arrière d'un processus ayant émis un message destiné à un autre groupe si celui-ci subit une défaillance, l'émetteur utilise un protocole à enregistrement de messages optimiste fondé sur l'émetteur. Lorsqu'une défaillance se produit, tous les processus des groupes ayant reçu un message orphelin du groupe fautif effectuent un retour arrière. Ce protocole a plusieurs inconvénients. D'une part, il ne confine les défaillances aux processus d'un seul groupe. D'autre part, si le nombre de messages inter-groupe augmente, le nombre de points de reprise forcés augmente également.

En se fondant sur cette notion de groupes, dans [47], les auteurs ont adapté un protocole de sauvegarde de points de reprise coordonnés afin d'éviter la sauvegarde simultanée de tous les points de reprise. Lorsque la sauvegarde des points de reprise est lancée, les processus sont divisés en groupes et les groupes sauvegardent les points de reprise tour à tour. Afin de garantir que les points de reprise forment un état global cohérent, les groupes qui ont sauvegardé leur point de reprise ne peuvent communiquer avec ceux qui ne l'ont pas encore fait.

2.2.4.2 Les protocoles à enregistrement de messages inter-groupe

Plusieurs autres études ont proposé des protocoles combinant un protocole de sauvegarde de points de reprise coordonnés et un protocole à enregistrement de messages.

Les messages entre les groupes sont enregistrés afin de limiter le nombre de groupes qui redémarrent [50] ou de confiner les défaillances à un seul groupe [51, 52, 49]. Tous ces travaux supposent que les applications sont déterministes par morceaux. Afin de garantir la cohérence de l'exécution, les auteurs de [49] ont montré que déterminants de tous les messages (intra et inter-groupe) doivent être sauvegardés.

Le premier protocole hiérarchique a été proposé dans [50]. Les auteurs ont proposé de combiner le protocole de sauvegarde de points de reprise coordonnés décrit dans [9] au protocole à enregistrement de messages optimistes [31]. Lorsqu'une défaillance se produit, la ligne de recouvrement est calculée de la même manière que dans [31] en tenant compte des messages enregistrés. Ainsi, la défaillance dans un groupe peut forcer le retour arrière de processus des autres groupes.

L'ensemble des protocoles décrits par la suite ont pour avantage de confiner les défaillances aux processus appartenant au même groupe que le processus fautif.

Dans les protocoles décrits dans [51] et dans [52], les messages inter-groupe sont enregistrés en utilisant un protocole à enregistrement de messages pessimiste fondé sur le récepteur [51] ou sur l'émetteur [52]. Au sein des groupes, le protocole de sauvegarde de points de reprise coordonnés utilisé est un protocole fondé sur le temps. Les déterminants des messages intra-groupe sont soit gérés de façon causale [51] soit pessimiste [52]. Lorsqu'une défaillance se produit, soit tous les processus appartenant au même groupe que le processus fautif redémarrent [52], soit seulement ceux qui doivent lui renvoyer un message [51]. Les messages inter-groupe sont rejoués sans forcer de retour arrière étant donné qu'ils sont enregistrés. Ces protocoles ont pour inconvénient de forcer la sauvegarde causale ou pessimiste des déterminants au sein des groupes et entre les groupes ce qui peut introduire un surcoût important comme expliqué dans le paragraphe précédent.

Afin de réduire le nombre des déterminants sauvegardés, dans le protocole décrit dans [49], les déterminants de tous les messages (inter et intra-groupe) sont enregistrés en utilisant la technique décrite dans [2]. Ce protocole a pour avantage de n'enregistrer qu'un sous-ensemble des déterminants grâce aux caractéristiques des communications MPI.

Le protocole décrit dans [48] ne précise aucune supposition sur le déterminisme des applications. L'enregistrement des messages inter-groupe est fondé sur l'émetteur. Lorsqu'une défaillance se produit, seuls les processus appartenant au même groupe que le processus fautif effectuent un retour arrière. Cependant, ce protocole ne sauvegarde aucune information sur les messages échangés au sein des groupes (aucun déterminant n'est enregistré) et ne fournit aucun autre mécanisme de gestion des causalités. De ce fait, seul un déterminisme total de l'application permet d'assurer que les messages orphelins seront réémis.

Première partie

Caractéristiques des applications du calcul haute performance

Chapitre 3

Déterminisme des communications dans les applications du calcul haute performance

Sommaire

3.1 Introduction 23
3.2 Standard MPI 24
3.2.1 Rappels sur quelques primitives MPI 24
3.2.2 Mise en œuvre des communications point à point dans MPI . 25
3.3 Modèle et définitions 26
3.3.1 Modèle 26
3.3.2 Définitions formelles 27
3.3.3 Application du modèle à MPI 27
3.4 Communications dans les applications MPI 27
3.4.1 Description des applications étudiées 27
3.4.2 Exemples de codes MPI 28
3.5 Analyse des applications 33
3.5.1 Résultats de l'analyse 34
3.6 Conclusion 35

3.1 Introduction

Les protocoles de tolérance aux fautes diffèrent, entre autre, par le déterminisme qu'ils supposent. En effet, les protocoles de sauvegarde de points de reprise supposent que les applications sont non déterministes, alors que les protocoles à enregistrement de messages supposent qu'elles sont déterministes par morceaux.

Dans ce chapitre, nous étudions le déterminisme des communications de 26 applications MPI afin de déterminer sa nature. Nous définissons une nouvelle classe de déterminisme que nous appelons "déterminisme des émissions" : pour les mêmes paramètres en entrée, chaque processus émet toujours la même séquence de messages d'une

exécution à une autre. Cependant, ces messages peuvent être reçus dans un ordre différent, sans pour autant avoir un quelconque impact sur les messages émis par la suite par ces processus. Nous montrons ici que la plupart de ces applications sont à émissions déterministes.

Ce chapitre est organisé comme suit. Le paragraphe 3.2 décrit quelques primitives de communication MPI, ainsi que la mise en œuvre des communications point à point. La paragraphe 3.3 présente le modèle et les définitions théoriques du déterminisme et du déterminisme des émissions. Par la suite, nous présentons le domaine d'application de chaque application étudiée dans le paragraphe 3.4. Puis, dans ce même paragraphe, nous exposons quelques exemples de schémas de communications déterministes, à émissions déterministes et non déterministes (paragraphe 3.4.2). Enfin, le paragraphe 3.5 est consacré à une analyse du déterminisme de 26 applications MPI représentatives du calcul haute performance.

3.2 Standard MPI

Le standard MPI [53] (*Message Passing Interface*) définit un interface pour la programmation par échange de messages incluant un ensemble de fonctions de communications collectives ou point à point. Cette interface est mise en œuvre dans plusieurs bibliothèques telles que OpenMPI [54] et MPICH [55].

Dans ce paragraphe, nous commençons par décrire quelques primitives de communication point à point dans MPI utilisées dans les exemples présentés dans le paragraphe 3.4.2. Par la suite, nous détaillons comment les évènements d'émission et de réception sont mis en œuvre.

3.2.1 Rappels sur quelques primitives MPI

Les communications point à point utilisent principalement deux fonctions : *MPI_Send* pour les émissions et *MPI_Receive* pour les réceptions. Dans la suite de ce paragraphe, seuls les prototypes des fonctions MPI en langage C seront donnés (les primitives en *FORTRAN* ont un paramètre supplémentaire *IEORR* pour les erreurs).

- *MPI_Send(void* buf, int nombre, MPI_Datatype type, int dest, int tag, MPI_Comm comm)* envoie le message contenu à l'adresse *buf*. Le tampon contient un nombre *nombre* d'éléments consécutifs de type *type*. Le message est envoyé au processus *dest* avec le tag *tag*.
- *MPI_Recv(void *buf, int nombre, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *statut)* reçoit un message contenant *nombre* éléments consécutifs de type *type*. Le message est stocké à l'adresse *buf*. Le message reçu vient du processus *source* avec le tag *tag*. Le *statut* contient les informations sur le message reçu. Notons que les paramètres *source* et *tag* peuvent ne pas être spécifiés. En d'autres termes, ils peuvent prendre la valeur *MPI_ANY_SOURCE* et *MPI_ANY_TAG* respectivement. Si ces deux valeurs sont utilisées dans un même *MPI_Recv*, n'importe quel message peut être reçu. Sinon, si le tag est positionné, tous les messages possédant ce tag peuvent être reçus et inversement

lorsque la source est positionnée.
Ces deux fonctions sont des fonctions bloquantes, c'est-à-dire qu'après avoir posté un *MPI_ Send* ou un *MPI_ Recv*, le processus est bloqué jusqu'à ce que le message soit émis ou reçu. Cependant, il existe des versions non bloquantes qui permettent aux processus d'avancer sans attendre la complétion de la fonction. Les deux principales primitives non bloquantes sont *MPI_ Isend* et *MPI_ Irecv*. Elles prennent les mêmes paramètres que les deux primitives bloquantes avec en plus le paramètre *MPI_ Request *requete* qui est utilisé par des primitives qui vérifient la complétion de ces opérations, par exemple *MPI_ Wait*, *MPI_ WaitAny*, *MPI_ Waitall*, ...

- *MPI_ Wait(MPI_ Request *requete, MPI_ Status *statut)* permet d'attendre la requête *requete*. Le paramètre *statut* permet de récupérer des informations sur la requête terminée (par exemple l'émetteur du message, la taille du message, . . .).
- *MPI_ Waitall(int nombre, MPI_ Request *tableau_ de_ requetes, MPI_ Status *tableau_ de_ statuts)* fonctionne de la même manière que *MPI_ Wait*, mais sur un ensemble de requêtes de nombre *nombre* répertoriées dans la table *tableau_ de_ requetes*. Elle retourne une table contenant le statut de chaque requête (terminaison avec succès ou erreur de la requête).
- *MPI_ Waitany(int compteur, MPI_ Request *tableau_ de_ requetes, int *index, MPI_ Status *statut)* permet d'attendre la complétion d'une requête parmi *compteur* requêtes. Ainsi, lorsqu'une seule requête est accomplie, l'exécution n'est plus en attente.
- *MPI_ Iprobe(int source, int tag, MPI_ Comm comm, int *flag, MPI_ Status *statut)* est la version non bloquante de la primitive *MPI_ Probe*. Elle permet de vérifier si un message correspondant aux paramètres en entrée est arrivé, sans le recevoir effectivement. Elle retourne $flag = vrai$ si le message est là. Le message identifié est le même que lors d'un appel à la fonction *MPI_ Recv(..., source, tag, comm, statut)*.

3.2.2 Mise en œuvre des communications point à point dans MPI

L'appel à une primitive d'émission ou de réception se compose de plusieurs évènements au niveau de l'application et au niveau de la bibliothèque. Nous distinguons ici deux évènements pour chaque communication : au niveau de l'application (l'appel à la primitive) et au niveau de la bibliothèque (l'arrivée ou le départ du message).

Afin de mieux différencier ces actions, nous définissons les 4 évènements suivants :

- *poster* : appel à la primitive d'émission
- *envoyer* : envoi du message par la bibliothèque
- *recevoir* : réception du message au niveau de la bibliothèque
- *delivrer* : réception du message par l'application

Les évènements d'émission et de réception se font comme suit :

- Émission : Le message est d'abord posté au niveau de l'application en faisant appelle à une primitive d'émission (*MPI_ Isend* par exemple). Puis la bibliothèque se charge de transférer effectivement les messages comme le montre la figure 3.1
- Réception : Le message est reçu au niveau de la bibliothèque. L'application poste

une requête pour recevoir le message. Celui-ci est délivré à l'application lorsque la requête est terminée, comme expliqué sur la figure 3.2.

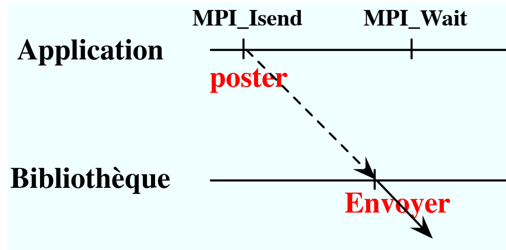


FIGURE 3.1 – Émission d'un message

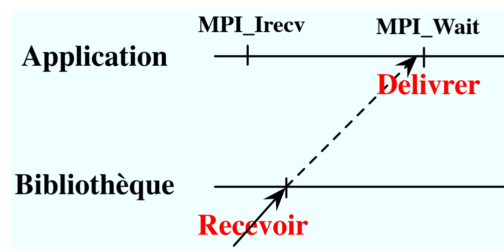


FIGURE 3.2 – Réception d'un message

3.3 Modèle et définitions

Dans ce paragraphe, nous commençons par présenter le modèle sur lequel nous nous fondons. Par la suite, nous donnons les définitions formelles du déterminisme et du déterminisme des émissions. Enfin, nous présentons comment le modèle peut être adapté à la bibliothèque MPI.

3.3.1 Modèle

Nous considérons des systèmes distribués asynchrones où une exécution parallèle est modélisée par un ensemble fini de processus et un ensemble fini de canaux reliant chaque paire de processus. Les canaux sont supposés fiables, FIFO, asynchrones et possédant des tampons de taille infinie.

Nous supposons un ensemble P de n processus. Chaque processus $p \in P$ est défini par un ensemble d'états C_p, un état initial c_0^p, un ensemble d'évènements V_p et une fonction de transition partielle :

$\mathscr{F}_p : C_p \times V_p \longrightarrow C_p$; si $p \neq q$ alors $V_p \cap V_q = \emptyset$.
Un état $c \in C_p$ est *final* s'il n'y a pas de transition valide à partir de c.

Un canal est modélisé par une file infinie. Un évènement d'envoi, $envoi(p, q, m) \in V_p$, ajoute un message m en queue du canal (p, q) ; un évènement de réception, $reception(q, p, m) \in V_q$, supprime le message m de la tête du canal (p, q).

Une exécution consiste en une séquence $E = e_1, e_2, \ldots$ d'évènements. Les évènements sont partiellement ordonnés via la relation de précédence de Lamport [56] notée $\rightarrow$. La configuration d'un système est représentée par un vecteur d'états $C = \{c_i, 1 \leq i \leq n, c_{i,j}, 1 \leq i, j \leq n\}$ où c_i représente l'état du processus i et $c_{i,j}$ l'état du canal (i, j). La séquence des configurations résultant de l'exécution E est définie comme suit :

- La configuration initiale du système est la configuration où tous les processus sont dans leur état initial et tous les canaux sont vides.
- Soit $C^i = \{c_p^i, c_{p,q}^i\}$ la configuration à l'étape i. Supposons que $e_i \in V_p$. Alors :

$$c_q^{i+1} = \begin{cases} c_q^i & \text{si } q \neq p \\ \mathscr{F}_p(c_p^i, e^i) & \text{si } q = p \end{cases}$$

- Une exécution est *correcte* si :
  - $\mathscr{F}_p(c_p^i, e^i)$ est définie à chaque étape
  - L'ordre causal entre les messages est respecté
  - Si l'évènement $reception(q, p, m)$ se produit alors le message m est en tête du canal (p, q).
  - Le dernier état de chaque processus est final.

$\mathscr{E}$ représente l'ensemble des exécutions correctes. Pour chaque exécution E, $E|p$ représente la sous séquence de E constituée d'évènement dans V_p. $C(E)|p$ représente la séquence d'états du processus p après chaque évènement dans $E|p$.

3.3.2 Définitions formelles

- **Définition 3.1** ***Déterminisme*** *Une exécution parallèle est déterministe si, pour chaque p, $C(E)|p$ est la même pour chaque $E \in \mathscr{E}$.*
- **Définition 3.2** ***Déterminisme des émissions*** *Une application est à émissions déterministes si, pour chaque p, $E|p$ contient la même séquence d'évènements $poster(p, q, m)$ pour chaque $E \in \mathscr{E}$.*

Notons que l'ensemble des applications déterministes est un sous-ensemble de l'ensemble des applications à émissions déterministes car ces dernières relâchent la conditions sur les réceptions.

3.3.3 Application du modèle à MPI

Comme expliqué dans le paragraphe 3.2.2, durant l'exécution d'une application MPI, deux évènements sont associés à chaque émission et réception de messages : un au niveau de l'application et un au niveau de la bibliothèque. Le déterminisme des émissions s'applique au niveau de l'application étant donné que c'est l'état de l'application que nous considérons pour les protocoles de recouvrement arrière. Cela signifie que le déterminisme des émissions s'applique aux évènements *"poster"* et *"delivrer"*. Ceci implique que la bibliothèque MPI elle même n'a pas besoin d'être à émissions déterministes.

3.4 Communications dans les applications MPI

Dans ce paragraphe, nous commençons par décrire brièvement les applications étudiées dans ce chapitre. Par la suite, nous présentons quelques exemples de schémas de communications déterministes, à émissions déterministes et non déterministes.

3.4.1 Description des applications étudiées

Les applications étudiées couvrent un large éventail de domaines scientifiques. Nous avons étudié des applications appartenant à quatre suites différentes : les *NAS Parallel*

Benchmarks [57], les *NERSC Benchmarks* [3], les *Sequoia Benchmarks* [6] et les *USQCD Benchmarks* [58].

Les applications *BT*, *CG*, *FT*, *LU*, *MG* et *SP* font partie de la suite des *NAS Parallel Benchmarks* et sont utilisées dans le domaine de la dynamique des fluides.

Les applications *CAM*, *GTC*, *IMPACT*, *MAESTRO*, *MILC* et *PARATEC* appartiennent à la suite des *NERSC Benchmarks*. Elles couvrent respectivement les domaines du climat, de l'énergie de fusion, des accélérateurs linéaires, de l'astrophysique, des jauges sur réseau et de la science des matériaux.

Les applications *AMG*, *IRS*, *lammp*, *Sphot* et *UMT* appartiennent à la suite des *Sequoia Benchmarks*. Elles sont utilisées en physique pour résoudre des équations de systèmes linéaires, la diffusion des rayonnements, la dynamique des molécules et la propagation des photons (Sphot et UMT). *IOR* est utilisée pour tester les performances des systèmes de fichiers.

L'application *CPS* fait partie de la suite des *USQCD Benchmarks*. Elle est utilisée dans le domaine de la chromodynamique quantique.

Nous avons également étudié des applications telles que SPECFEM3D [59] qui est utilisée pour simuler des tremblements de terre, ainsi que RAY2MESH [60] qui est également utilisée en géophysique.

Enfin, nous avons aussi analysé un noyau *Nbody* (prédiction du mouvement des objets célestes) et un noyau *Jacobi* (résolution de systèmes d'équations linéaires).

Afin de classifier ces applications, les auteurs de [61] ont proposé un classement non pas par domaines d'application, mais par noyaux de calcul utilisés par les applications. Ils ont défini treize *Motifs* dont sept représentent les principales méthodes numériques utilisées dans les applications scientifiques : algèbre linéaire dense, algèbre linéaire creuse, méthodes spectrales, méthodes N-Corps, grilles structurées, grilles non structurées et *MapReduce*. Dans ce qui suit, nous ne nous intéresserons pas aux applications de type *MapReduce* car les protocoles de tolérance aux fautes utilisés dans ce type d'applications ne correspondent pas aux protocoles étudiés ici. Le tableau 3.1 résume la classification de quelques unes des applications étudiées ici.

Algèbre Linéaire Dense	Algèbre Linéaire Creuse	Méthodes Spectrales	Simulations N-Corps	Grilles Structurées	Grilles Non Structurées
BT, LU, PARATEC	CG, MAESTRO, MILC	FT, IMPACT, MILC, PARATEC, SPECFEM3D	GTC, LAMMPS, MILC, Nbody	CAM, GTC, IMPACT, MAESTRO, MG, PARATEC, SPECFEM3D	MAESTRO

Table 3.1 – Classification des applications selon les Motifs

3.4.2 Exemples de codes MPI

Nous présentons ici quelques exemples de schémas de communications rencontrés dans les applications décrites dans le paragraphe 3.4.1 selon la nature du déterminisme

de ces derniers.

3.4.2.1 Déterminisme des communications

Un schéma de communications est dit déterministe si les mêmes messages sont émis et reçus dans le même ordre à chaque exécution.

Un exemple simple de schéma de communications déterministe est présenté ci-dessous. Dans cet exemple, le processus envoie une donnée aux autres processus en utilisant des communications bloquantes et des réceptions nommées.

```
if(rank == 0)
  {
   for ( i = 1 ; i < nbprocs ; i++)
     MPI_Send(T[i], ..., i, ...);
   /*  où i est le récepteur du message */
  }
else
  {
   MPI_Recv(x,..., 0, ...);
/* où 0 est l'émetteur du message */
  }
```

Un autre exemple avec des appels à des primitives non bloquantes est présenté dans la portion de code ci-dessous :

```
for(i=0; i<nb; i++)
{
   MPI_Irecv(T[i],..., i, ...);
/*  où i est l'émetteur du message */
   MPI_Isend(x, ...,  i, ...);
/* où i est le récepteur du message */
   MPI_Wait( ...);
   MPI_Wait( ...);
}
```

Dans cet exemple, pour chaque couple (émetteur, récepteur), chaque message doit être émis et reçu avant de passer au couple suivant, assurant ainsi que l'ordre de réception et d'émission des messages est toujours respecté.

L'ensemble des portions de code décrites ci-dessus sont extraites de l'application *MAESTRO*.

L'exemple de schéma de communications précédent reste déterministe si la source du message au niveau de la réception est inconnue :

```
for(i=0; i<nb; i++)
{
   MPI_Irecv(T[i],..., MPI_ANY_SOURCE, i, ...);
/*  où i est l'émetteur du message */
   MPI_Send(x, ...,  i, id, ...);
/* où i est le récepteur du message */
   MPI_Wait( ...);
}
```

En effet, dans cet exemple, chaque message a une étiquette (le tag) différente, ce qui permet d'identifier les messages et ainsi de les recevoir dans le bon ordre. Ainsi, une source non définie n'est pas toujours une cause de non déterminisme.

Observons le code ci-dessous :

```
for(i=0; i<nb_recv; i++)
   MPI_Irecv(T[i], ,i..,...);
for(i=0; i<nb_send; i++)
   MPI_Send( ..., i, ...);
MPI_Waitall(nb_recv, ...)
```

Plusieurs primitives d'émission et de réception sont postées, puis un *MPI_ Waitall* est posté afin d'attendre la complétion de l'ensemble des émissions et des réceptions. Les messages reçus peuvent être reçus dans un ordre différent, mais seront délivrés à l'application dans le même ordre. Étant donné que nous considérons les évènement *delivrer* pour déterminer la nature du déterminisme, ce schéma est également déterministe.

Ce schéma reste déterministe même avec une source non nommée (*MPI_ ANY_ SOURCE*) :

```
for(i=0; i<nb_recv; i++)
    MPI_Irecv(R[i], ...,
      MPI_ANY_SOURCE, tag[i]...);
for(i=0; i<nb_send, i++)
    MPI_Isend(S[i], ..., proc,
      tag[i], ...);
mpi_Waitall(...)
```

où $tag[i]$ est l'étiquette de la communication. Dans cette application, une étiquette différente est attribuée à chaque émetteur, permettant d'éviter de confondre les messages. Ce code est extrait de l'application *MILC*.

Le schéma ci-dessous est similaire aux deux schémas précédents mais il contient *MPI_ Waitany* au lieu de *MPI_ WaitAll*. On le retrouve, par exemple, dans les applications *lammps* et *IRS* de la suite des *Sequoia Benchmark*.

```
for(i=0; i<nb_recv; i++)
   MPI_Irecv(T[i], ,i..,...);
for(i=0; i<nb_send; i++)
   MPI_Send( ..., i, ...);
for(i=0; nb_recv; i++)
  {
   MPI_Waitany(nb_recv, , &i, );
   /*calcul sur T[i]*/
  }
```

Dans cet exemple, les messages peuvent être délivrés dans un ordre différent d'une exécution à une autre (car la primitive *MPI_ Waitany* attend la complétion de n'importe quelle requête). Après la réception de tous les messages (à la sortie de la boucle contenant le *MPI_ Waitany*) le résultat est le même.

Afin de différencier ce type de schémas de communications d'un schéma non déterministe où les réceptions et les émissions peuvent se faire dans un ordre différents, nous définissons un nouveau type de déterminisme appelé **"déterminisme des émissions"**.

3.4.2.2 Déterminisme des émissions

Un schéma de communication est dit à émissions déterministes, si pour un même ensemble de données en entrée et pour chaque processus, la séquence des émissions est toujours la même. Ceci implique que du côté du récepteur, l'ordre des réceptions n'a pas d'effet sur la suite de l'exécution.

L'exemple ci-dessous est également à émissions déterministes. La source et le *tag* ne sont pas définis (*MPI_ ANY_ SOURCE* et *MPI_ ANY_ TAG*), comme c'est le cas dans l'application *IOR* de la suite des *Sequoia Benchmark*

```
if(rank==0)
  for(i=0; i<nb; i++)
     {
      MPI_Recv(hostname, ...,
      ANY_SOURCE, ANY_TAG,...);
      if(hostname == localhost)
      count ++;
     }
else MPI_Send(localhost, ..., 0, 0, ...);
```

Cette portion de code calcule le nombre de processus s'exécutant sur le même nœud (ce nœud héberge le processus de rang 0). L'ordre de réception des messages n'a pas d'impact sur la valeur finale du compteur. Cependant, le nombre de processus qui sont sur le même nœud que le processus 0 doit être déterministe pour un ensemble de données en entrée et de paramètres d'exécution donnés.

Un autre exemple est décrit ci-dessous :

```
if(rank ==0)
  for(i=1; i<nb_procs; i++)
   {
    MPI_Recv(T[i], ..., ANY_SOURCE, ANY_TAG,...);
   }
else MPI_Send(T[rank], ..., 0, 0, ...);
MPI_Barrier(...);
sort(T);
```

Dans ce cas, il n'y a aucun moyen de garantir que le contenu du tableau T après l'appel à *MPI_Barrier()* est le même pour différentes exécutions. Cependant, avant que l'utilisation du tableau n'influence les processus du système, il est trié. L'exemple est considéré à émissions déterministes car :

1. Le résultat n'est jamais envoyé
2. Juste après la boucle, T est trié, assurant ainsi que le résultat sera toujours le même pour le même ensemble de paramètres d'entrée.

3.4.2.3 Non déterminisme

Un schéma de communications est considéré comme non déterministe si il n'est ni déterministe ni à émissions déterministes. En d'autres termes, un schéma de communications est non déterministe si les messages peuvent être émis dans un ordre différent à chaque exécution. Le schéma qui suit est un exemple de schéma de communications non déterministe. Il est extrait de l'application *AMG* de la suite des *Sequoia Benchmarks*.

```
for(i=0; i<n; i++)
  MPI_Irecv(... , tag2, ...)
for(i=0; i<n; i++)
  MPI_Isend(..., tag1, ...)
...
while()
 {
  MPI_IProbe(MPI_ANY_SOURCE, tag1,
  flag, status)
  while(flag ==true)
    {
     proc = &status.mpi_SOURCE;
     MPI_Recv(x, ..., proc, tag1, ...);
     /*modifier x*/
     MPI_Send(x, ..., proc, tag2, ...);
    }
 }
```

Le non déterminisme est dû à l'utilisation de *MPI_Iprobe* avec une source inconnue (*MPI_ANY_SOURCE*). Le processus teste l'arrivée d'un message de n'importe quelle

source, reçoit le message puis envoie des données dépendantes des données reçues. Ainsi, les messages émis dépendent des messages reçus, et les messages peuvent ne pas être reçus dans le même ordre d'une exécution à une autre, les messages peuvent donc ne pas être émis dans le même ordre d'une exécution à une autre. La condition de déterminisme des émissions n'est donc plus respectée.

Un autre exemple de schéma de communication non déterministe est retrouvé dans la plupart des applications maître/travailleurs. L'exemple présenté ci-dessous est extrait de la version maître/travailleurs de l'application *Ray2Mesh*.

```
if(id == root)
 for(i=1; i<nb_procs; i++)
 {
  MPI_Recv(x,..., MPI_ANY_SOURCE, .., &status);
  MPI_Send(y, ..., status.MPI_SOURCE, ...);
 }
else
  {
   MPI_Send(x, ..., root, ...);
   MPI_Recv(y, ..., root, ...);
  }
```

Ce code implémente un ordonnancement "premier arrivé premier servi" où les travailleurs récupèrent des tâches du maître. Un travailleur plus rapide aura plus de tâches. Une fois que le processus maître reçoit une requête d'un travailleur, il lui envoie une tâche. Ainsi, d'une exécution à une autre, le processus maître peut ne pas émettre la même séquence de messages (destinataires différents, contenus différents), étant donné que le paramètre source, du côté du maître, est mis à *MPI_ANY_SOURCE*.

3.5 Analyse des applications

À partir des définitions décrites ci-dessus, nous avons analysé 26 applications de calcul haute performance. L'analyse a été effectuée de manière statique : à partir des définitions formelles du déterminisme et du déterminisme des émissions, nous avons analysé chaque portion de code contenant des communications. Puis, les applications ont été classifiées comme suit :

- Une application est déterministe si et seulement si elle contient uniquement des schémas déterministes
- Une application est à émissions déterministes si et seulement si elle contient au moins un schéma où les émissions sont déterministes et aucun schéma non déterministe
- Une application est non déterministe si elle contient au moins un schéma de communications non déterministe

3.5.1 Résultats de l'analyse

Le tableau 3.2, présente, pour chaque application, les différentes classes de schémas de communications utilisés et le nombre de schémas de chaque classe rencontrés dans l'application. Concernant les primitives collectives, étant donné que leur déterminisme dépend de leur mise en œuvre dans la bibliothèque MPI, nous les avons considérées comme déterministes. Afin d'évaluer le nombre d'appels aux différentes primitives MPI, nous avons simplement compté le nombre de fois où chaque primitive est appelée. Autrement dit, si une primitive est appelée dans une boucle, nous ne comptons qu'un seul appel. Il est en de même si une primitive est appelée dans un bloc conditionnel.

La colonne “Classification” donne la classe à laquelle chaque application appartient : “D” pour déterministe, “ED” pour émissions déterministes et “ND” pour non déterministe.

À partir de cette table, nous constatons que seulement deux applications sont non déterministes (*Sequoia AMG*, et la version maître/travailleurs de *Ray2mesh*). Nous remarquons également que plusieurs applications sont à émissions déterministe. En y incluant l'ensemble des applications déterministes, nous remarquons que presque toutes les applications sont à émissions déterministes.

Application	Déterministes	Émissions déterministes	Non déterministes	Classification
NAS-BT	5	0	0	D
NAS-CG	10	0	0	D
NAS-DT	5	0	0	D
NAS-EP	0	0	0	D
NAS-FT	0	0	0	D
NAS-LU	12	0	0	D
NAS-MG	1	0	0	D
NAS-SP	7	0	0	D
NERSC-CAM	65	0	0	D
NERSC-GTC	3	0	0	D
NERSC-IMPACT	127	0	0	D
NERSC-MAESTRO	18	0	0	D
NERSC-MILC	628	0	0	D
NERSC-PARATEC	4	0	0	D
Sequoia-UMT	2	0	0	D
Sequoia-lammp	35	2	0	ED
Sequoi-IOR	0	2	0	ED
Sequoia-AMG	4	76	1	ND
Sequoia-Sphot	8	0	0	D
Sequoia-IRS	5	2	0	ED
USQCD-CPS	31	0	0	D
Jacoby	2	0	0	D
Nbody	1	0	0	D
Ray2mesh	1	1	0	ED
Ray2mesh-MS	1	2	3	ND
SpecFEM3D	1	0	0	D

TABLE 3.2 – Déterminisme des applications MPI

3.6 Conclusion

Au cours de ce chapitre, nous avons présenté quelques exemples de schémas de codes présents dans les applications MPI du calcul haute performance. Nous avons classé ces schémas selon leur déterminisme et avons mis en avant une nouvelle classe de déterminisme : le “déterminisme des émissions”. Une application est dite à émissions déterministes si, pour un même ensemble de données en entrée, les mêmes messages sont toujours émis dans le même ordre par chaque processus. Du côté des récepteurs, l’ordre de réception des messages n’a pas d’effet sur les messages émis.

Nous avons formellement défini le déterminisme et le déterminisme des émissions. En se fondant sur ces définitions, nous avons analysé 26 applications MPI du calcul haute performance et avons conclu que la plupart d’entre elles sont à émissions déterministes.

La présence du déterminisme des émissions nous permet d’utiliser ses caractéristiques pour proposer de nouveaux protocoles de tolérance aux fautes où il n’est pas nécessaire de forcer le redémarrage de tous les processus et où il n’est pas nécessaire d’enregistrer tous les déterminants car l’ordre de réception des messages n’est pas important. Dans la seconde partie de ce document, nous proposons deux protocoles de tolérance aux fautes fondés sur le déterminisme des émissions.

CHAPITRE 4

Les schémas de communication dans les applications du calcul haute performance

Sommaire

4.1 Introduction **37**
4.2 Étude des schémas de communications dans les applications MPI **38**
4.2.1 Les études de schémas de communications existantes 38
4.2.2 Schémas de communications des primitives de communications collectives 39
4.2.3 Schémas de communications dans les applications MPI 40
4.3 Partitionnement **41**
4.3.1 Les approches possibles 42
4.3.2 Partitionnement fondé sur la bissection 42
4.4 Fonctions de coût **44**
4.4.1 Coût de l'enregistrement des messages α 45
4.4.2 Coût du redémarrage β 45
4.4.3 Volume des messages enregistrés L et nombre processus à redémarrer R 45
4.5 Évaluation **47**
4.6 Conclusion **48**

4.1 Introduction

Les protocoles de tolérance aux fautes hiérarchiques sont principalement utilisés en créant des groupes de processus, comme expliqué dans le Chapitre 2. Ces groupes peuvent être fondés sur les schémas de communications des processus par exemple. Dans ce chapitre, nous étudions l'existence de groupes de processus dans ces schémas.

Dans plusieurs applications MPI, les processus communiquent plus avec certains processus qu'avec d'autres. En s'appuyant sur cette caractéristique, nous montrons qu'il est possible de définir des groupes de processus malgré la présence de primitives de communications collectives où tous les processus sont impliqués. Les groupes sont créés de façon à minimiser leur taille et à minimiser le volume des messages échangés entre eux. Ces deux paramètres sont contradictoires, car plus grande est la taille du groupe, plus petit est le volume des messages inter-groupe, et inversement. L'évaluation présentée ici montre qu'il est possible de créer des groupes de processus selon le coût de la gestion des défaillances. En effet, si l'on considère que les messages inter-groupe sont enregistrés et que les processus d'un même groupe redémarrent, si l'enregistrement des messages est plus coûteux que le redémarrage, avoir de grands groupes de processus serait plus adapté et inversement.

Les travaux présentés dans ce chapitre ont été réalisés en collaboration avec Bora Uçar, chercheur à l'École Normale Supérieure de Lyon et Esteban Meneses, doctorant à *UIUC* (États-Unis).

Ce chapitre est organisé comme suit. Le paragraphe 4.2 présente une étude des schémas de communications. Nous commençons par décrire les études existantes dans le paragraphe 4.2.1. Puis, nous présentons les schémas de communications des primitives collectives dans MPICH2 (paragraphe 4.2.2) et de certaines applications MPI (paragraphe 4.2.3). Le paragraphe 4.3 décrit l'algorithme de partitionnement que nous avons utilisé pour créer les groupes. Enfin, le paragraphe 4.5 présente les résultats de différents regroupements possibles.

4.2 Étude des schémas de communications dans les applications MPI

Ce paragraphe présente, dans un premier temps, les études des schémas de communications existant dans la littérature. Puis, nous présentons les schémas de communications des primitives de communications collectives dans MPICH2 ainsi que les schémas de communications obtenus en exécutant les applications décrites dans le Chapitre 3. Ces schémas ont été obtenus en modifiant le code de MPICH2 [1] afin de collecter les informations sur les communications. L'ensemble des applications ont été exécutées sur la grappe de Rennes de Grid'5000 ayant les caractéristiques décrites dans le paragraphe 5.4.2 du Chapitre 5.

4.2.1 Les études de schémas de communications existantes

Les schémas de communications de la plupart des applications étudiées ici ont déjà été présentés. Dans [3], les auteurs ont exposé les caractéristiques de communications des *NERSC Benchmarks* (schémas de communications, nombre d'appels aux primitives MPI, ...). Dans [4], les auteurs ont présenté les schémas des communications des *NAS Parallel Benchmarks*. Cependant, les schémas présentés dans [4] ne prennent

1. http ://svn.mcs.anl.gov/repos/mpi/mpich2/trunk :r7592

pas en considération les algorithmes utilisés par les primitives collectives. Par exemple, la représentation d'un *MPI_Broadcast* initié par le processus de rang 0 montre que ce processus a communiqué avec tous les autres processus. L'étude des schémas de communications de primitives de communications collectives présentée dans ce chapitre montre que ces primitives ne sont pas mises en œuvre de cette manière [62].

Comme expliqué dans les études cités ci-dessus, la plupart des applications utilisent des primitives de communications collectives. Dans [63], les auteurs ont montré que dans plusieurs applications, les communications collectives utilisent de petits messages, et ce indépendamment de la taille du problème.

D'autre part, dans [64], les auteurs ont montré que, pour plusieurs applications MPI du calcul haute performance, les processus communiquent plus avec certains processus qu'avec d'autres.

Ces deux derniers points indiquent que des groupes de processus peuvent être créés, et que la présence de communications collectives n'aura pas un grand impact sur le volume des données échangées entre les groupes.

4.2.2 Schémas de communications des primitives de communications collectives

Nous nous intéressons aux schémas de communications des primitives collectives car les schémas générés par celles-ci peuvent être difficiles à partitionner, tous les processus étant impliqués.

Les algorithmes utilisés pour la mise en œuvre des primitives de communications collectives dépendent principalement de la taille des messages et du nombre de processus impliqués (en général, puissance de deux ou pas). Nous ne présentons ici que les schémas de communications des primitives de communications collectives selon la taille des messages, car la plupart des applications étudiées ont été exécutées sur un nombre de processus en puissance de deux. La table 4.1 présente les algorithmes utilisés par chaque primitive de communications collectives dans MPICH2 selon la taille des messages. Ces algorithmes sont détaillés dans [62]. Les Figures 4.1 et 4.2 présentent, pour chaque algorithme mentionné dans la table 4.1, le volume des données échangés entre chaque processus (émetteur en abscisse et récepteur en ordonnée). Toutes les primitives ont été exécutées sur 64 processus. Pour les tailles des messages, nous avons choisi 4 octets pour les petits messages, 16 Ko pour les messages de taille moyenne et 2 MO pour les grands messages. L'échelle utilisée pour les algorithmes utilisés pour de grands messages est logarithmique. La Figure 4.1 présente les schémas de communications des différents algorithmes utilisés par primitive *MPI_Alltoall* (et de la collectives *MPI_Reduce_scatter* pour les grands messages) présentés dans la table 4.1. La Figure 4.2 présente les schémas de communications des algorithmes utilisés par les autres primitives de communications collectives (les schémas de communications des algorithme *recursive doubling* et *recursive halving* étant identiques, nous ne présentons que le schémas de l'algorithme *recursive doubling*).

Les schémas présentés sur les figures 4.2(e) et (f), montrent que des groupes de processus existent naturellement (des groupes de 8 ou 16 processus). Les schémas pré-

sentés par les figures 4.2(d), 4.2(g) et 4.2(h) montrent un regroupement moins naturel. Néanmoins, des groupes peuvent être créés facilement dans ces cas aussi (par exemple des groupes de 8 processus).

L'ensemble des schémas présentés sur la figure 4.1 sont, quant à eux, difficiles à partitionner manuellement en groupes de processus. Cependant, comme expliqué dans le paragraphe 4.2.1, les communications collectives utilisent principalement des messages de petite taille, qui correspondent à la Figure 4.1(a), dont l'aspect ne correspond pas à celui où tous les processus communiquent avec tous les processus (contrairement aux figure 4.1(b) et 4.1(c)). De plus, sur des machines de grande taille, utiliser des primitives où tous les processus communiquent avec tous les processus ne passera pas bien à l'échelle.

Collective	Petits Messages	Messages Moyens	Grands Messages
MPI_Allgather	Recursive Doubling ($<$ 512 Ko)		Ring ($\geq$ 512 Ko)
MPI_Allreduce	Recursive Doubling ($\leq$ 2 Ko)		Rabenseifner($>$ 2Ko)
MPI_Alltoall	Bruck ($\leq$ 256 Octets)	irecv-isend ($\leq$ 32 Ko)	Pairwise exchange ($>$ 32 Ko)
MPI_Broadcast	Arbre Binomial ($<$ 12 Ko)		Van de Geijn ($\geq$ 12 Ko)
MPI_Gather	Arbre Binomial		Arbre Binomial
MPI_Reduce	Arbre Binormal ($\leq$ 2 Ko)		Rabenseifner ($>$ 2Ko)
MPI_Reduce_scatter	Recursive Halving ($<$ 512 Ko)		Pairwise exchange ($\geq$ 512 Ko)
MPI_Scatter	Arbre Binomial		Arbre Binomial

TABLE 4.1 – Algorithmes utilisés pour les communications collectives dans MPICH2

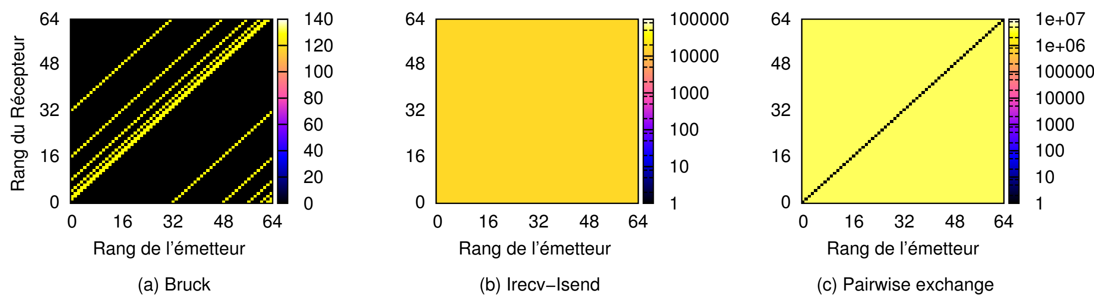


FIGURE 4.1 – Schémas de communication de la primitive *MPI_Alltoall*

4.2.3 Schémas de communications dans les applications MPI

Dans le paragraphe précédent (4.2.2), nous avons montré que, mis à part les schémas de communications de la primitive *MPI_Alltoall*, les schémas de communications des primitives de communications collectives sont tels qu'il semble possible de former des groupes de processus. Nous montrons dans ce paragraphe que certaines communications point à point peuvent engendrer des schémas de communication difficile à partitionner, si bien que le nombre de groupes et la taille de chacun ne peuvent être connus à priori. La figure 4.4 présente les schémas de communications de plusieurs applications décrites

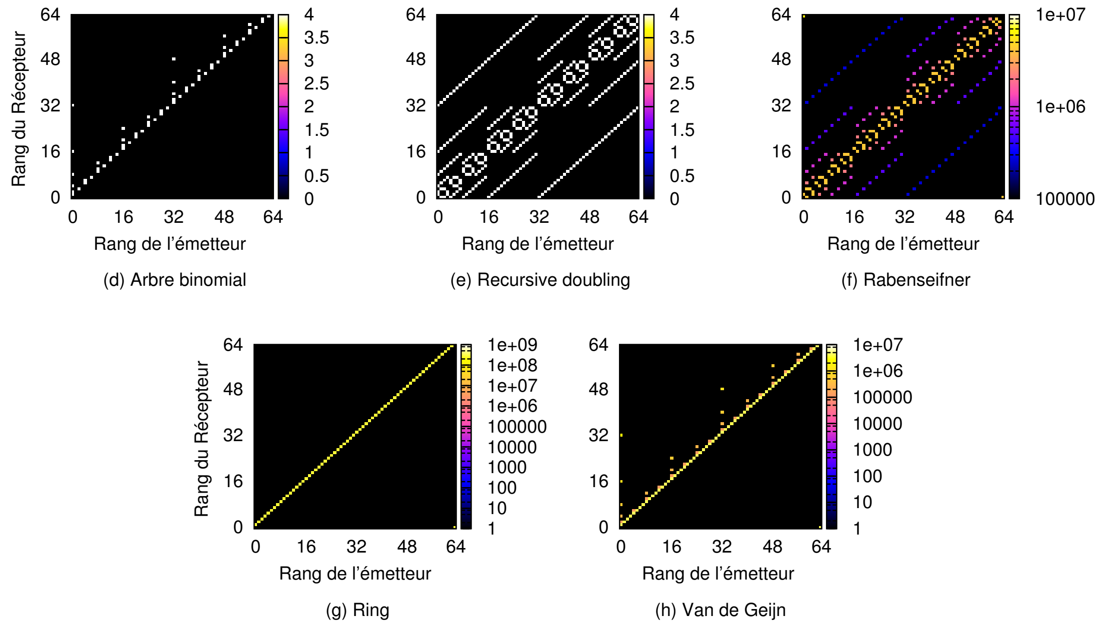


FIGURE 4.2 – Schémas de communication des primitives collectives dans MPICH2

dans le Chapitre 3. Toutes ces applications ont été exécutées sur 1024 processus à l'exception de *GTC* (figure 4.4(g)) et *Maestro* (figure 4.4(i)) qui ont été exécutées sur 512 processus. Pour plus de clarté, pour chaque application, la figure présente son schéma de communications et un zoom sur les 128 premiers processus afin de mieux voir l'aspect du schéma. Le volume des messages, pour l'ensemble des figures, est représenté en échelle logarithmique.

Nous remarquons que pour certaines applications, par exemple *CG* (figure 4.4(b)), les schémas de communications sont tels qu'il est possible de créer des groupes de processus manuellement (32 processus par groupe pour *CG*).

Pour d'autres applications telles que FT (figure 4.4(c)) , Maestro (figure 4.4(i)) et Paratec (figure 4.4(k)), les processus ne sont pas regroupés naturellement, ce qui rend le partitionnement manuel plus compliqué. L'utilisation d'outils automatiques devient donc, dans certains cas, indispensable. Dans les paragraphes suivants, nous montrons que même pour ces applications, il est possible de former des groupes de processus en minimisant la taille des groupes et le volume des messages inter-groupe.

4.3 Partitionnement

Nous proposons un algorithme de partitionnement fondé sur la bissection pour la création automatique de groupes dans le cadre des protocoles à enregistrement partiel de messages. L'objectif de la création des groupes est de limiter les communications inter-groupe, d'augmenter le nombre de groupes et de limiter la taille maximale des

groupes. Nous présentons d'abord les limites des solutions existantes puis décrivons notre méthode.

4.3.1 Les approches possibles

Une version simplifiée du problème de partitionnement, dont le but est de minimiser la taille des messages enregistrés et de maximiser la taille des parties, correspond au problème NP-complet de partitionnement de graphe (voir problème ND14 dans [65]). Ceci peut être remarqué en considérant un graphe dont les sommets représentent les processus et les arêtes les communications entre deux processus. L'utilisation des heuristiques de partitionnement de graphe nécessite de connaître le nombre maximum de partie. Ceci est possible mais nécessite néanmoins une étude de l'application et de l'architecture de la machine visée.

Une variante commune de ce problème spécifie le nombre de parties (en d'autres termes, spécifie la taille moyenne de la partie) et force les parties à avoir des tailles proches (réduisant ainsi la taille maximale d'une partition). Le problème demeure NP-complet [66]. Des outils tels que MeTiS [67] et Scotch [68] peuvent être utilisés pour résoudre ce problème. Une manière possible mais non économique d'utiliser ces outils est de partitionner les processus avec différents nombres de groupes (2, 4, 8, ...) et de choisir la meilleure partition rencontrée.

Dans [48], les auteurs ont proposé un algorithme qui partitionne les processus en un nombre de groupes non fourni en entrée. La seule contrainte qu'ils imposent est la taille maximale des groupes. L'algorithme consiste à ajouter, dans un groupe, des processus qui ont communiqué avec les processus de ce groupe tant que la taille limite n'est pas atteinte. Ces processus sont choisis selon l'ordre décroissant du volume des données échangées. L'algorithme décrit ici est fondé sur la bissection.

4.3.2 Partitionnement fondé sur la bissection

Les algorithmes de partitionnement par bissection sont utilisés de manière récursives dans les outil de partitionnement de graphes et hypergraphes (par exemple PaToH [69], MeTiS et Scotch) afin de partitionner un graphe ou un hypergrpahe donné en un nombre donné de parties. Cette approche fonctionne comme suit : pour un nombre donné de K parties, la bissection divise le graphe ou hypergraphe original en deux parties de taille quasiment égale, puis partitionne récursivement les deux parties en $K/2$ parties jusqu'à atteindre le nombre désiré de parties.

Nous avons adapté l'approche par bissection au problème de partitionnement. L'algorithme proposé est décrit sur la figure 4.3. L'algorithme prend en entrée un ensemble de P processus et une matrice M de taille $P \times P$ représentant les communications entre les processus où $M(u, v)$ est le volume des données envoyés du processus u au processus v. L'algorithme retourne le nombre de parties $K^\star$ et la partition $\Pi^\star = \langle P_1, P_2, \ldots, P_K^\star \rangle$. Dans l'algorithme, l'opération $\Pi \leftarrow \Pi \ominus P_k \oplus P_{k_1} \oplus P_{k_2}$ supprime la parties P_k de la partition Π et la remplace par P_{k_1} et P_{k_2} comme nouvelles parties, où $P_k = P_{k_1} \cup P_{k_2}$.

Entrée: Ensemble des processus $P = \{p_1, p_2, \ldots, p_P\}$; la matrice M de taille $P \times P$ représentant les communications entre les processus
Sortie: $K^\star$: le nombre de parties ; $\Pi^\star = \langle P_1, P_2, \ldots, P_K^\star \rangle$: une partition des processus
1: $K^\star \leftarrow K \leftarrow 1$; $B^\star \leftarrow B \leftarrow 0$
2: $\Pi^\star \leftarrow \Pi \leftarrow \langle P_1 = \{p_1, p_2, \ldots, p_P\} \rangle$ /* une partie unique */
3: **tant que** il y a une partie à analyser **faire**
4: Soit P_k la plus grande partie
5: **si** DevraitPartitionner(P_k) **alors**
6: $\langle P_{k_1}, P_{k_2} \rangle \leftarrow$Bissect($P_k, M(P_k, P_k)$)
7: **si** AccepteBissection(P_{k_1}, P_{k_2}) **alors**
8: $K \leftarrow K + 1$
9: $\Pi \leftarrow \Pi \ominus P_k \oplus P_{k_1} \oplus P_{k_2}$ /* remplacement de P_k */
10: $B \leftarrow B + \sum M(P_{k_1}, P_{k_2})$ /* Mise à jour du volume des messages inter-partie */
11: **si** Meilleure(Π) **alors**
12: $\Pi^\star \leftarrow \Pi$; $B^\star \leftarrow B$; $K^\star \leftarrow K$ /* sauvegarder le résultat */
13: **sinon**
14: marquer P_k afin de ne plus l'analyser dans les lignes 3 and 4.
15: **retourner** $\Pi^\star$

Figure 4.3 – L'algorithme de partitionnement proposé

$M(U, V)$ représente les messages envoyés des processus de l'ensemble U vers les processus de l'ensemble V.

L'algorithme utilise la bissection pour partitionner le nombre de processus en un nombre inconnu de parties. Initialement, tous les processus sont dans une seule partie. À chaque étape, la partie la plus grande est partitionnée, si elle doit l'être, en deux (par la routine Bissect), puis, si la bissection est acceptable (cette notion est déterminée par la routine AccepteBissection), elle est remplacée par les deux parties résultant de la bissection. Si la nouvelle partition est la meilleure jusqu'à présent (le test est fait par la routine Meilleur), elle est sauvegardée comme étant un résultat possible. Puis l'algorithme prend la nouvelle plus grande partie et refait les étapes décrites ci-dessus.

Soit P_k la plus grande partie. Si P_k doit être partitionnée (déterminé par la fonction DevraitPartitionner), elle est divisée en deux (P_{k_1} et P_{k_2}), puis un test vérifie si la nouvelle partition est acceptable. Si elle ne l'est pas, la bissection est annulée et P_k reste inchangée tout au long de l'algorithme. Si, au contraire, elle est acceptée, P_k est partitionnée pour de bon : une fois qu'un test Meilleur retourne vrai (pour la partition P_k), P_k est définitivement remplacée dans $\Pi^\star$.

Le cœur de l'algorithme est la routine Bissect. Cette routine prend en entrée un ensemble de processus et les communications entre eux et essaye de partitionner l'ensemble des processus en deux parties de taille quasiment égale en utilisant un outil de partitionnement existant. MeTis ou Scotch peuvent être utilisés si les communications sont bidirectionnelles ($M(u, v) \neq 0 \implies M(v, u) \neq 0$), dans quel cas $\frac{M+M^T}{2}$ peut être

utilisé. PaToH peut être utilisé si chaque communication (bidirectionnelle ou pas) peut être représentée par une arête unique.

La routine MEILLEUR nécessite une fonction de coût afin d'évaluer la partition. Cette fonction doit être définie selon les métriques que l'utilisateur veut optimiser. Elle est fonction de la taille des parties et du volume des messages inter-partie.

La routine DEVRAITPARTITIONNER est utilisée afin d'arrêter le partitionnement des petites parties. Elle retourne faux si la taille de la partie en question est plus petite qu'un seuil. Le seuil que nous avons utilisé a été fixé à 1. Avec un seuil plus grand, l'algorithme serait plus rapide.

Enfin la routine ACCEPTEBISSECTION retourne vrai dans les deux cas suivants :

1. Π et $\Pi' = \Pi \ominus P_k \oplus P_{k_1} \oplus P_{k_2}$, $\text{cout}(\Pi') \leq \text{cout}(\Pi)$ selon la fonction de coût.
2. Si cette condition n'est pas vérifiée, un meilleur partitionnement peut toujours être obtenu avec plus de bissections. Pour cela, nous avons adapté la formule du calcul de la force d'un graphe [70]. Pour une partition $\Pi = \langle P_1, P_2, \ldots, P_K \rangle$, nous utilisons $\text{force}(\Pi) = \frac{B}{K-1}$ comme force, B étant le volume des données inter-partie. La bissection est acceptée si $\text{force}(\Pi') \leq \text{force}(\Pi)$. Si la bissection réduit la force, elle peut être bénéfique, car elle incrémente la taille des données inter-partie de peu et respecte le nombre de parties.

4.4 Fonctions de coût

Afin d'exécuter l'algorithme de bissection, une fonction de coût doit d'abord être définie. Celle-ci dépend du protocole étudié.

Étant donné que nous utilisons ce protocole dans le cadre des protocoles de tolérance aux fautes, les métriques que nous considérons sont liés à la quantité de ressources perdues due à la gestion des défaillance. La fonction définie ici ne donne pas une évaluation exacte du protocole étudié, car plusieurs paramètres devraient être pris en compte (occupation mémoire par exemple).

Le principal objectif de cette fonction de coût est de trouver un compromis entre le nombre de processus à redémarrer et la quantité de données à enregistrer.

Le coût est modélisé, pour un partitionnement donné, comme fonction du volume total de messages à enregistrer et du nombre de processus à redémarrer. La formule utilisée est de la forme suivante :

$$\text{cout}(\Pi) = \alpha \times L + \beta \times R \tag{4.1}$$

où L représente le ratio des données enregistrées et R le ratio des processus à redémarrer après une défaillance. Ils dépendant du protocole utilisé. En effet, selon le nombre de messages enregistrés, le nombre de processus à redémarrer change. Les paramètre α et β représentent respectivement le coût associé à l'enregistrement des messages et celui associé au redémarrage après une défaillance.

4.4.1 Coût de l'enregistrement des messages α

En se fondant sur les résultats que nous présentons dans le paragraphe 4.5, nous fixons la valeur de α à 23% car celle-ci offre un meilleur compromis entre la taille des groupes et le volume de données entre les groupes. Ainsi $\alpha = 23\%$.

4.4.2 Coût du redémarrage β

Le coût du redémarrage β représente le temps perdu après une défaillance. Il dépend du moment où elle se produit. Dans [71], les auteurs ont proposé une formule afin d'évaluer l'intervalle optimum de sauvegarde des points de reprise (I). Les paramètres nécessaires au calcul de I sont [72] :

- Le temps moyen entre les défaillances ($MTBF$)
- Le temps nécessaire afin de sauvegarder les points de reprise de tous les processus (C)
- Le temps de redémarrage de toute l'application R

La formule suppose que tous les processus sauvegardent leur point de reprise en même temps. Nous avons utilisé ce même paramètre même si, dans les protocoles que nous considérons, seuls quelques processus effectuent un retour arrière.

L'intervalle de sauvegarde des points de reprise est : $I = \sqrt{(2 \times C \times (MTBF + R))}$.

Si nous supposons que les défaillances sont uniformément distribuées sur un intervalle, en moyenne, une défaillance se produit à l'instant $\frac{I}{2}$. Le temps total perdu sera égal à la durée des calculs perdus ($I/2$) en plus du temps de redémarrage (R). Les valeurs des paramètres sont fixées selon [73] :

- $MTBF = 1$ jour
- $C = 30$ minutes
- $R = 30$ minutes

L'intervalle de sauvegarde des points de reprise est donc égal à $I = 297$ minutes. Le temps de calcul perdu par $MTBF$ est égal à 179 ce qui représente $12,4\%$ du $MTBF$. Nous considérons donc que $\beta = 12,4\%$.

4.4.3 Volume des messages enregistrés L et nombre processus à redémarrer R

Comme dit précédemment, le calcul de L et de R dépend du protocole étudié. Dans ce document, nous présentons deux protocoles. Nous décrivons ici brièvement leur fonctionnement afin pour pouvoir calculer L et R.

1. Un protocole de sauvegarde de points de reprise pour les applications à émissions déterministes : Dans ce protocole, chaque groupe se voit attribuer un identifiant qu'on appelle *époque*. Les messages sont enregistrés s'ils vont d'un groupe vers un groupe avec une époque supérieure. Lorsqu'une défaillance se produit dans un groupe, tous les processus des groupes avec une époque supérieure effectuent un retour arrière.

2. HydEE : Dans HydEE, tous les messages inter-groupe sont enregistrés. Lorsqu'une défaillance se produit, seul les processus du groupe fautif effectuent un retour arrière.

4.4.3.1 Fonction de coût pour le premier protocole

1. Calcul de L

   Afin de calculer L, nous avons supposé que la taille des communications entre les groupes est la même dans les deux directions. Soit B la taille totale des messages inter-groupe et D le volume total des messages.

$$L = \frac{B}{2 \times D}$$

2. Calcul de R

   Afin d'évaluer le nombre moyen de processus à redémarrer (R), soient N le nombre de processus, K le nombre de groupes numérotés de 1 à K (selon les époques) et T_i la taille du groupe i. Nous supposons que si le processus d'un groupe fait un retour arrière, tous les processus du même groupe effectuent également un retour arrière.

   Dans ce protocole, lorsqu'un processus d'un groupe subit une défaillance, tous les processus avec une époque supérieure effectuent un retour arrière. Ainsi, si un processus du groupe i subit une défaillance, $\sum_{j=i}^{K} T_j$ effectuent un retour arrière. La probabilité qu'une défaillance se produise dans le groupe i est $\frac{T_i}{N}$.

   Donc sur les K groupes, le nombre moyen de processus à redémarrer est :

   $\frac{T_1}{N} \times \sum_{j=1}^{K} T_j + \frac{T_2}{N} \times \sum_{j=2}^{K} T_j + ... + \frac{T_k}{N} \times T_K$

   La valeur de R, qui correspond au ratio des processus à redémarrer est donc égal à :

$$R = \frac{1}{N^2}(\sum_{i=1}^{K} T_k^2 + \sum_{i=1}^{K-1}(T_i * \sum_{j=i+1}^{K-1} T_j))$$

   Étant donnée une partition $\Pi = \langle P_1, P_2, \ldots, P_K \rangle$ la formule utilisée est de la forme suivante :

$$\text{cout}(\Pi) = 23\% \times \frac{B}{2 \times D} + 12.4\% \times R \tag{4.2}$$

4.4.3.2 Fonction de coût pour HydEE

Dans ce protocole, tous les messages inter-groupe sont enregistrés. La valeur de L est donc égale à la taille totale des messages inter-groupe.

Afin de calculer le nombre moyen de processus à redémarrer (R), soit N le nombre de processus, K le nombre de groupes numérotés de 1 à K et T_i la taille du groupe

i. Si un processus du groupe i subit une défaillance, T_i processus redémarrent. La probabilité qu'une défaillance se produise dans le groupe i est de $\frac{T_i}{N}$. Le nombre de processus du groupe i à redémarrer en cas de défaillance est $T_i \times \frac{T_i}{N}$. Sur l'ensemble des groupes, le nombre moyen de processus à redémarrer est : $\sum_i (T_i \times \frac{T_i}{N})$. Considérons N exécutions où une seule défaillance se produit pour chaque exécution, les défaillances étant uniformément distribuées sur les processus. Le nombre moyen de processus à redémarrer sur les N exécutions est donc égal à
$\frac{\sum_i (T_i \times \frac{T_i}{N})}{N}$. Ainsi

$$R = \frac{\sum_i T_i^2}{N^2}$$

Étant donnée une partition $\Pi = \langle P_1, P_2, \ldots, P_K \rangle$ qui implique l'enregistrement d'un volume de données L, la formule utilisée est de la forme suivante :

$$\text{cout}(\Pi) = 23\% \times \frac{L}{D} + 12.4\% \times \frac{\sum_k |P_k|^2}{P^2} \qquad (4.3)$$

où $D = \sum \sum (M(u, v))$ est le volume total des messages.

4.5 Évaluation

Ce paragraphe présente différents résultats de regroupement obtenus en utilisant l'outil décrit dans le paragraphe 4.3.

Le but de cette évaluation est de montrer qu'il est possible de former des groupes de processus selon les contraintes imposées (petits groupes, faible volume de messages inter-groupe), et ce malgré l'aspect des schémas de communications de certaines applications (MAETRO (figure 4.4(i), Paratec (figure 4.4(k)) et FT (figure 4.4(c)) et malgré la présence de la primitive *MPI_Alltoall* dans certains codes.

La table 4.2 montre, pour chaque application, le pourcentage du nombre de processus par groupe et le volume des messages inter-groupe si le coût de l'enregistrement des messages est faible. La table 4.3 montre les mêmes données si ce coût est élevé. Afin d'obtenir ces résultats, les valeurs du paramètre α de la fonction de coût sont respectivement 7% et 23%. Les applications ont été exécutées sur 1024 processus à l'exception de *MAESTRO* et *GTC* qui ont été exécutées sur 512 processus. Nous avons réalisé ces deux mesures afin de montrer qu'il est possible de créer des groupes de processus selon le coût attribué à chaque paramètre. Comme expliqué précédemment, en supposant que messages inter-groupe sont enregistrés et que tous les processus d'un groupe redémarrent en cas de défaillance dans ce groupe, si l'objectif est de minimiser le nombre de processus à redémarrer, il faut maximiser le volume des données inter-groupe et inversement.

Nous remarquons que dans les deux cas, il est toujours possible de trouver un regroupement qui respecte les conditions choisies. Pour des groupes de petite taille, les groupes créés contiennent moins de 7% du total des processus avec moins de 35 % de messages inter-groupe pour la plupart des applications. Pour des groupes de grande taille, le volume des données inter-groupe ne dépasse pas les 22%. Dans les deux cas, *FT* donne de

Applications	Nb Groupes	% Taille Groupe	% Volume Inter-groupe
NPB BT	19	5.66%	22.23%
NPB CG	32	3.125%	16.23%
NPB FT	32	3.125%	97%
NPB LU	16	6.25%	9.68%
NPB MG	16	6.25%	26.49%
NPB SP	21	5.27%	23.56%
GTC	32	3.125%	11.02%
LAMMPS	32	3.125%	19.83%
MAESTRO	16	6.25%	34.63%
NBODY	32	3.125%	3.42%
PARATEC	31	3.32%	14%
SPECFEM3D	32	3.125%	16.56%

Table 4.2 – Petits groupes

Applications	Nb Groupes	% Taille Groupe	% Volume Inter-groupe
NPB BT	8	12.5%	16.92%
NPB CG	32	3.125%	16.23%
NPB FT	2	50%	50%
NPB LU	16	6.25%	9.68%
NPB MG	4	25%	12.14%
NPB SP	8	12.5%	16.14%
GTC	16	6.25%	8.01%
LAMMPS	7	15.65%	9.39%
MAESTRO	5	21.87%	21.63%
NBODY	32	3.125%	3.42%
PARATEC	16	6.25%	10.69%
SPECFEM3D	14	7.80%	11.12%

Table 4.3 – Grands groupes

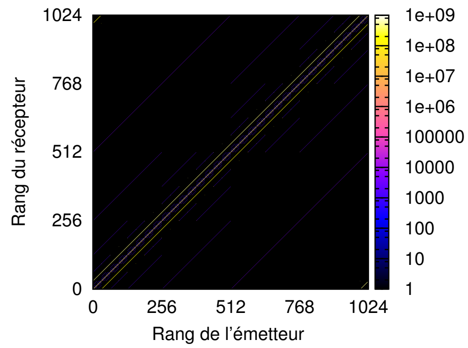
moins bons résultats car celle-ci utilise principalement la primitive *MPI_ Alltoall*. Nous remarquons également que pour des applications où les groupes de processus existent naturellement (par exemple CG), la taille des groupes et le volume des données inter-groupe est stable.

4.6 Conclusion

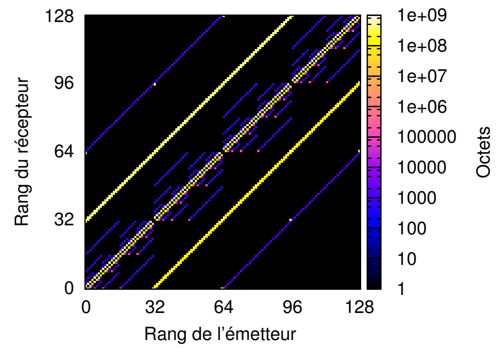
Au cours de ce chapitre, nous avons présenté les schémas de communications des différents algorithmes utilisés par les primitives de communications collectives dans MPICH2. Nous avons remarqué que malgré la participation de l'ensemble des processus dans ce type de communications, les schémas créés par celles-ci permettent de former des groupes de processus mis à part le schéma engendré par les algorithmes utilisés par la primitive *MPI_ Alltoall*.

Nous avons également présenté les schémas de communications de plusieurs applications MPI du calcul haute performance. Nous avons remarqué que pour certaines applications, les processus forment naturellement des groupes. Pour d'autres applications, malgré l'absence ou la faible présence de la primitive *MPI_ Alltoall*, la formation de groupes de processus n'est pas triviale. Nous avons proposé par la suite un algorithme de partitionnement grâce auquel il est possible de créer des groupes de processus pour la plupart des applications selon les paramètres que l'on désire respecter (taille des groupes et volumes des messages inter-groupe).

Dans la seconde partie de ce document, nous proposons des protocoles hiérarchiques pour les applications à émissions déterministes qui tirent bénéfice de cet outil de partitionnement.

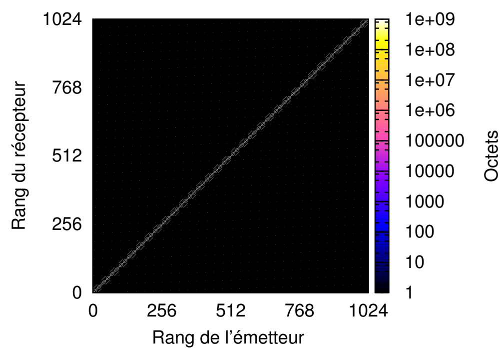


(a) Schéma

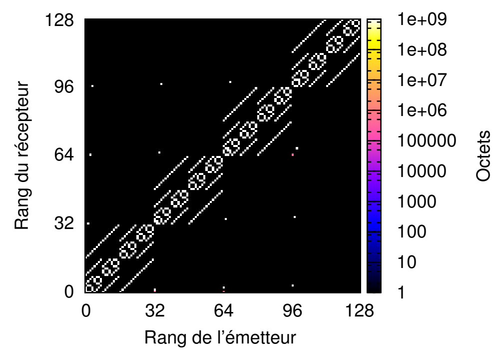


(b) Zoom

(a) BT

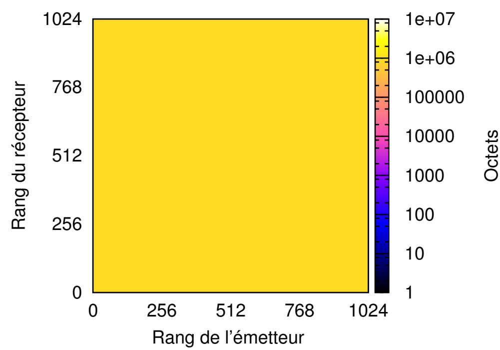


(a) Schéma

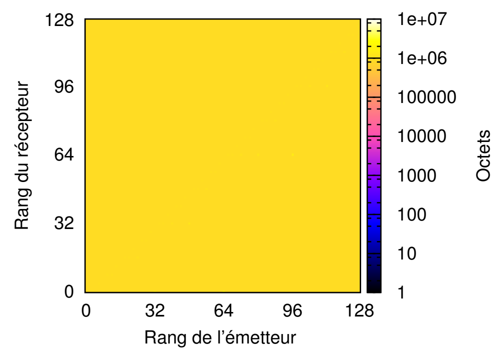


(b) Zoom

(b) CG

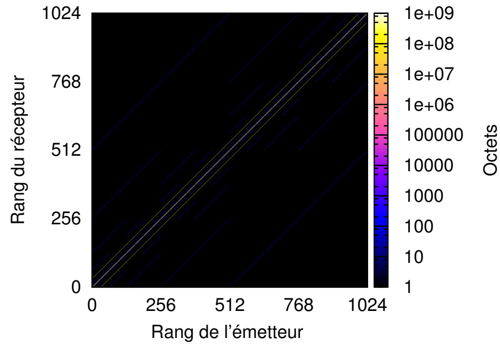


(a) Schéma

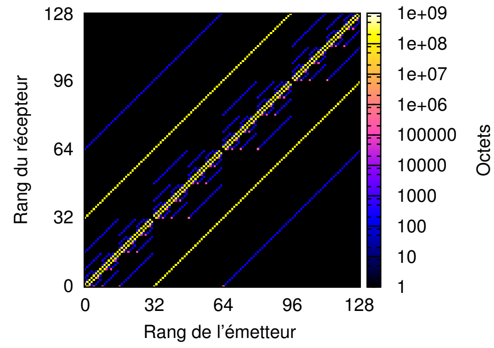


(b) Zoom

(c) FT

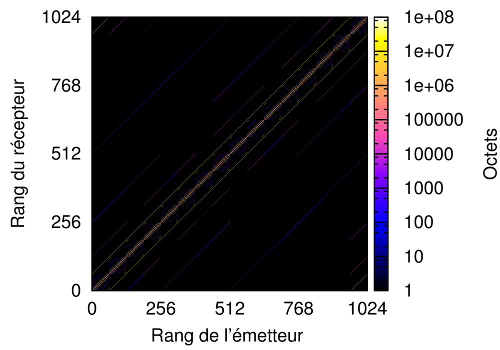


(a) Schéma

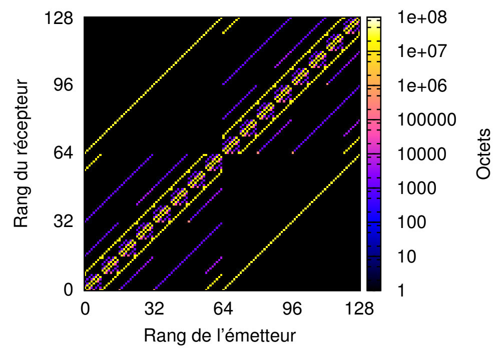


(b) Zoom

(d) LU

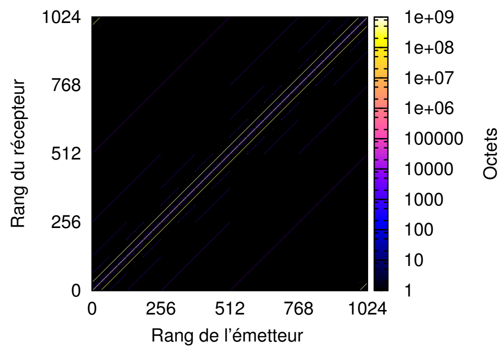


(a) Schéma

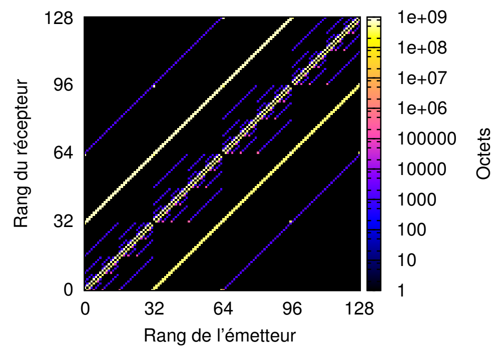


(b) Zoom

(e) MG

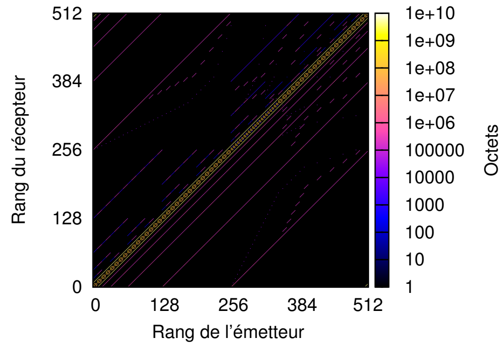


(a) SP

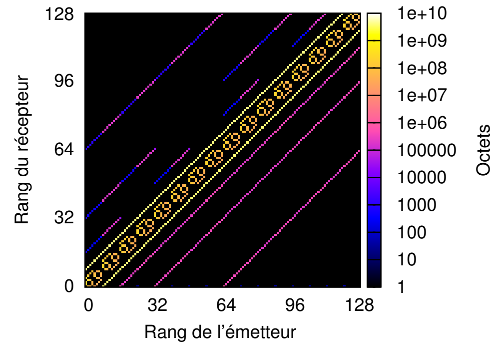


(b) Zoom

(f) SP

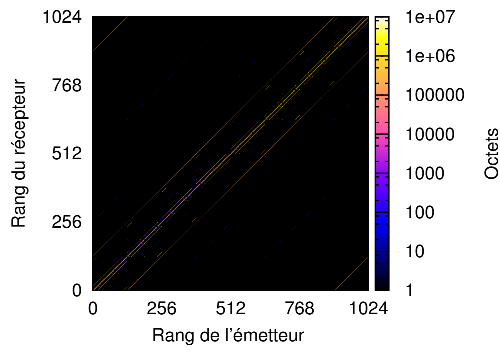


(a) Schéma

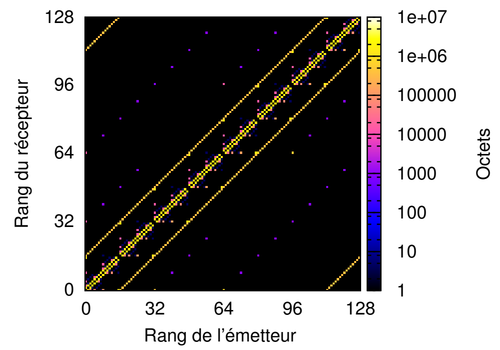


(b) Zoom

(g) GTC

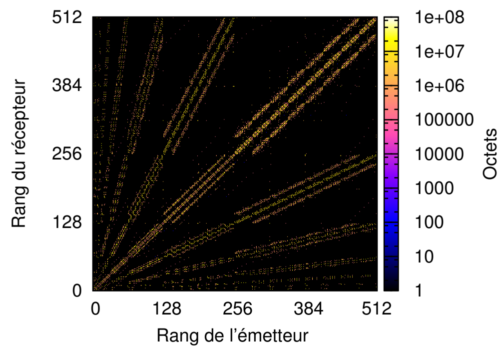


(a) Schéma

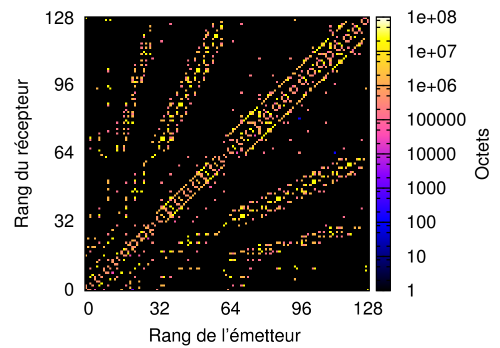


(b) Zoom

(h) Lammps

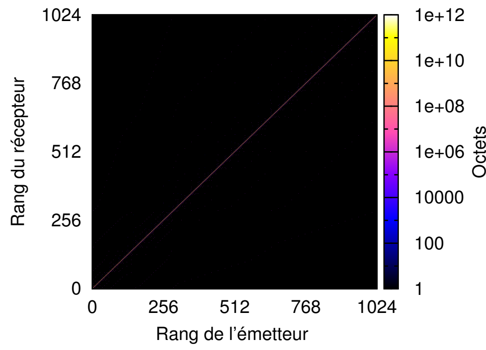


(a) Schéma

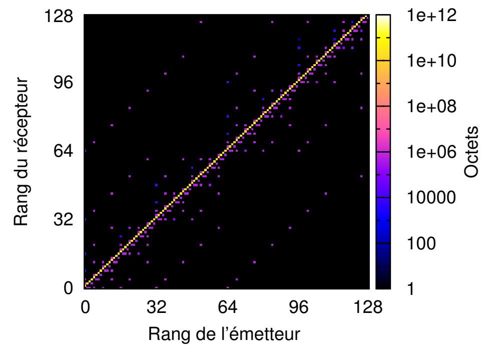


(b) Zoom

(i) Maestro

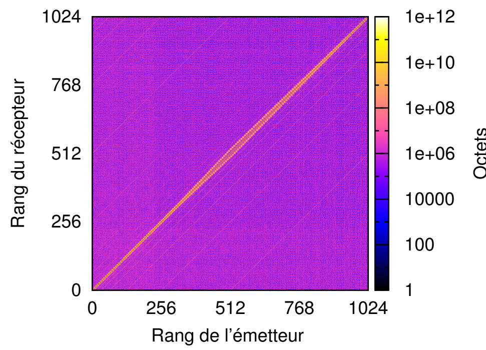


(a) Schéma

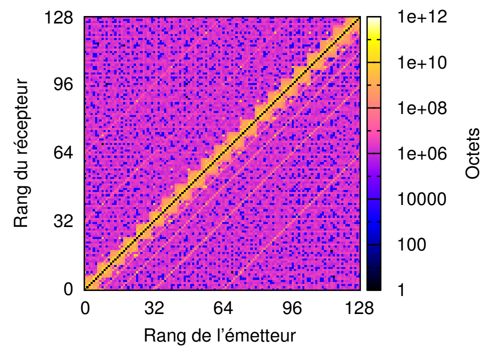


(b) Zoom

(j) Nbody

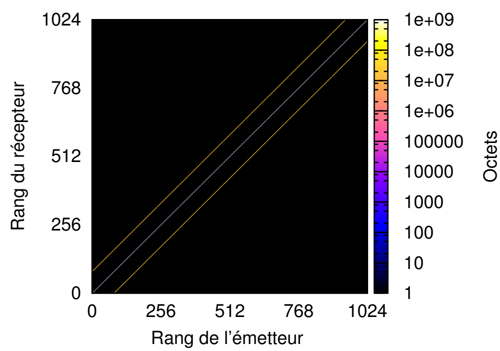


(a) Schéma

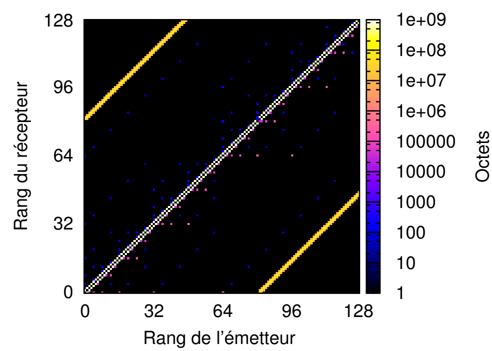


(b) Zoom

(k) Paratec



(a) Schéma



(b) Zoom

(l) SpecFem3D

FIGURE 4.4 – Schémas de communications des applications MPI

Deuxième partie

Protocoles de tolérance aux fautes pour les applications à émissions déterministes

Chapitre 5

Un protocole de sauvegarde de points de reprise non coordonnés pour les applications à émissions déterministes

Sommaire

5.1 Introduction **56**

5.2 Contexte **56**

5.2.1 Modèle 56

5.2.2 Déterminisme des émissions et tolérance aux fautes 56

5.2.3 Effet domino dans les protocoles fondés sur le déterminisme des émissions 57

5.3 Description **58**

5.3.1 Fonctionnement normal 58

5.3.2 Gestion des défaillances 59

5.3.3 Gestion des causalités 59

5.3.4 Pseudo code 62

5.3.5 Suppression des données obsolètes 66

5.3.6 Preuve 66

5.4 Évaluation **70**

5.4.1 Description du prototype 70

5.4.2 Plate-forme d'expérimentation 72

5.4.3 Évaluation des performances en fonctionnement normal . . . 73

5.4.4 Évaluation du nombre de processus à redémarrer : 75

5.4.5 Utilisation du protocole en définissant des groupes de processus 76

5.5 Conclusion **80**

5.1 Introduction

Les protocoles de sauvegarde de points de reprise non coordonnés souffrent de l'effet domino car ils supposent que les applications sont non déterministes, forçant ainsi le retour arrière des processus ayant reçu des messages orphelins.

Ce chapitre présente un protocole de sauvegarde de points de reprise non coordonnés pour les applications à émissions déterministes qui évite l'effet domino. Il est combiné à un protocole à enregistrement de messages fondé sur l'émetteur, mais contrairement aux protocoles à enregistrement de messages existants, celui-ci ne nécessite la sauvegarde que d'un sous-ensemble de messages avec leur numéro d'ordre d'émission mais d'aucun numéro d'ordre de réception.

L'utilisation du protocole sur des groupes de processus montre qu'il est possible de ne faire redémarrer qu'un sous-ensemble de processus en cas de défaillance en n'enregistrant qu'un sous-ensemble de messages.

Ce chapitre est organisé comme suit. Nous commençons par décrire, dans le paragraphe 5.2, l'impact du déterminisme des émissions sur les protocoles de tolérance aux fautes. Par la suite, nous exposons le fonctionnement du protocole dans le paragraphe 5.3. Ce paragraphe contient également les algorithmes utilisés dans le protocole et la suppression des données obsolète ainsi qu'une preuve formelle sur la validité de l'exécution après une défaillance. Le paragraphe 5.4 présente l'ensemble de l'évaluation des performances en fonctionnement normal ainsi qu'une évaluation du protocole sur des groupes de processus en utilisant l'outil décrit dans le Chapitre 4.

5.2 Contexte

Dans ce paragraphe, nous présentons le modèle de fautes sur lequel le protocole est fondé. Par la suite, nous expliquons comment le déterminisme des émissions peut être utilisé dans les protocoles de tolérance aux fautes. Enfin, nous décrivons l'effet domino pouvant exister dans des protocoles fondés sur le déterminisme des émissions.

5.2.1 Modèle

Nous nous fondons sur le modèle décrit dans le Chapitre 3. Par souci de simplicité, nous notons $envoi(m)$ et $reception(m)$ les évènements d'envoi ($poster(p, q, m)$) et de réceptions ($reception(p, q, m)$) décrits dans le Chapitre 3. Les pannes considérées sont franches et multiples.

5.2.2 Déterminisme des émissions et tolérance aux fautes

Le déterminisme des émissions garantit que si un processus reçoit des messages dans un ordre différent d'une exécution correcte à une autre, il émet toujours la même séquence de messages. Ainsi, après une défaillance, les protocoles fondés sur le déterminisme des émissions sont capables de rejouer les messages orphelins sans rejouer les messages dont ils dépendent dans le même ordre que lors de l'exécution sans fautes. De

tels messages ne sont donc pas une source d'incohérence dans des protocoles fondés sur le déterminisme des émissions contrairement aux protocoles existants. Sur la figure 5.1, si le processus p_2 subit une défaillance (et effectue un retour arrière vers son dernier point de reprise), les messages m_1 et m_2 sont nécessaires à son redémarrage. Cependant, étant donné que ces deux messages ne sont pas causalement dépendants, quel que soit l'ordre de leur réception, le message m_3 sera émis. Ces messages deviennent ainsi des doublons que nous appelons "messages orphelins" ou "messages dupliqués".

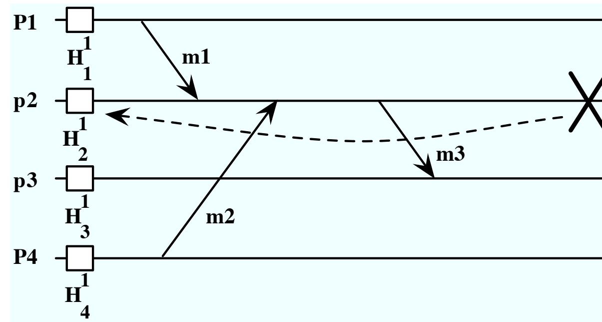


FIGURE 5.1 – Déterminisme des émissions

Le fait que les messages orphelins soient toujours réémis permet d'éviter de forcer le retour arrière des processus ayant reçus de tels messages, évitant ainsi l'effet domino décrit dans le Chapitre 2. De plus, étant donné que l'ordre de réception des messages n'a pas d'impact sur les messages émis par la suite, il n'est pas nécessaire d'enregistrer les déterminants. Sur la figure 5.1, après la défaillance de p_2, le processus p_3 n'effectue pas de retour arrière pour recevoir m_3 et les déterminants des messages m_1 et m_2 ne sont pas enregistrés pour garantir leur ordre de réception.

5.2.3 Effet domino dans les protocoles fondés sur le déterminisme des émissions

Le déterminisme des émissions permet d'éviter le retour arrière des processus ayant reçu des messages orphelins en cas de défaillance. Cependant, étant donné que le processus fautif a besoin des messages envoyés en exécution normale pour sa réexécution, si le protocole utilisé n'est pas fondé sur l'enregistrement des messages, les processus ayant émis ces messages doivent effectuer un retour arrière. Ceci peut amener tous les processus à effectuer des retours arrière, provoquant ainsi un effet domino. Sur la figure 5.2, si le processus p_2 subit une défaillance, il effectue un retour arrière vers son dernier point de reprise. Afin d'émettre le message m_3, le processus p_1 doit revenir à son premier point de reprise H_1^1. Pour pouvoir redémarrer, p_1 a besoin du message m_2, le processus p_2 fait donc un nouveau retour arrière vers H_2^1. Pour les mêmes raisons, p_3 effectue un retour arrière vers le point de reprise H_3^1. Ainsi, la défaillance du processus p_2 provoque un effet domino.

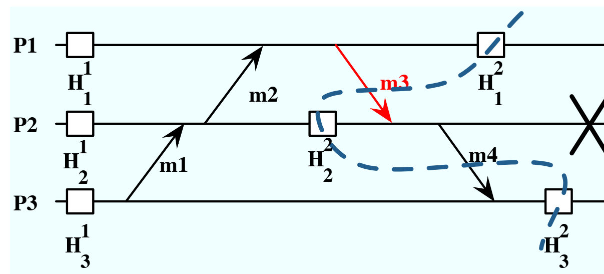


FIGURE 5.2 – Effet Domino

5.3 Description

Dans ce paragraphe, nous présentons un protocole de sauvegarde de points de reprise non coordonnés pour les applications à émissions déterministes. Nous expliquons comment le protocole évite l'effet domino décrit dans le paragraphe 5.2.3 en n'enregistrant qu'un sous-ensemble des messages.

Nous commençons par décrire, dans un premier temps, le fonctionnement normal du protocole. Par la suite, nous décrivons son comportement après une défaillance et expliquons comment les causalités sont gérées. Par la suite, nous donnons la preuve que l'exécution après une défaillance reste correcte. Enfin, nous présentons le pseudo-code du protocole.

5.3.1 Fonctionnement normal

Le protocole étant un protocole de sauvegarde de points de reprise non coordonnés, les processus sauvegardent leurs points de reprise indépendamment les uns des autres. Des époques, qui correspondent aux numéros des points de reprise, sont définies sur chaque processus. Chaque époque est incrémentée à chaque sauvegarde de point de reprise. Afin d'éviter l'effet domino décrit dans le paragraphe 5.2.3, l'ensemble des messages qui vont du "passé vers le futur" sont enregistrés, en d'autres termes, les messages envoyés à une époque E_e et reçus dans une époque E_r telle que $E_e < E_r$. Le protocole à enregistrement de messages utilisé est un protocole fondé sur l'émetteur. Sur la figure 5.2, le message m_3 est enregistré car il va de l'époque 1 à l'époque 2. Afin de détecter l'époque du récepteur, ce dernier envoie un acquittement, contenant son époque, à l'émetteur. Lorsque celui-ci reçoit cet acquittement, il compare son époque à celle contenu dans l'acquittement et enregistre le message si nécessaire.

Lorsque le processus prend son point de reprise, toutes les informations nécessaires (décrites dans le paragraphe 5.3.4) ainsi que les messages enregistrés sont sauvegardés sur support stable.

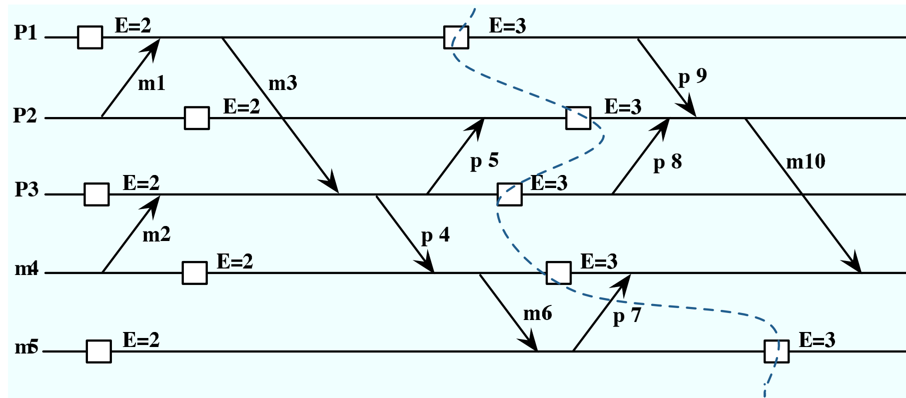


FIGURE 5.3 – Scénario d'exécution

5.3.2 Gestion des défaillances

Lorsqu'une défaillance se produit, le processus défaillant p fait un retour arrière vers son dernier point de reprise sauvegardé. Puis, chaque processus qui a envoyé un message au processus p qui a été reçu après ce point de reprise, fait également un retour arrière vers le point de reprise précédent l'émission du message. Par exemple, Sur la figure 5.3, si le processus p_2 subit une défaillance après la réception du message m_9, les processus p_1 et p_3 doivent effectuer un retour arrière pour renvoyer les messages m_9 et m_8.

Les processus ayant envoyé un message aux processus qui font un retour arrière font également un retour arrière sous les mêmes conditions. Sur la figure 5.3, si le processus p_4 subit une défaillance après la réception du message m_{10}, le processus p_2 doit effectuer un retour arrière pour renvoyer m_{10}. Puis les processus p_1 et p_3 doivent faire de même pour renvoyer les messages m_9 et m_8 respectivement. Cependant, étant donné que le message m_7 est envoyé de l'époque 2 à l'époque 3, il est enregistré, le processus p_5 n'a donc pas besoin de faire de retour arrière pour le renvoyer.

Afin de ne pas confondre les messages orphelins avec de nouveaux messages à recevoir, les processus filtrent ces messages en se fondant sur leur date. Cette opération est détaillée dans le paragraphe 5.3.4

5.3.3 Gestion des causalités

Même si les messages peuvent être reçus dans n'importe quel ordre, les évènements restent partiellement ordonnés sur un processus. Il est par exemple possible d'ordonner la réception d'un message et l'émission d'un message qui suit. Durant le redémarrage, l'ordre causal doit être assuré afin de garantir que les messages orphelins sont réémis. Étant donné que le protocole ne fait pas redémarrer les processus qui ont reçu des messages orphelins et que les messages enregistrés sont renvoyés sans conditions, des messages qui sont causalement dépendants peuvent être émis au même moment durant le redémarrage et délivrés dans un ordre qui ne respecte pas l'ordre causal, menant

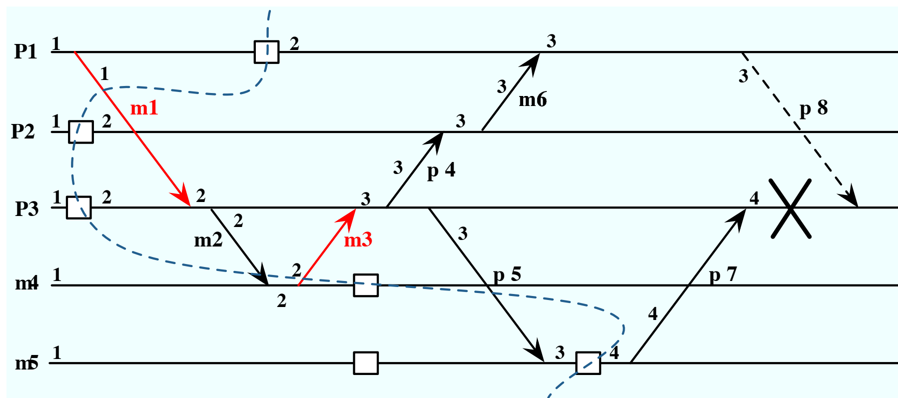


FIGURE 5.4 – Problèmes des causalités

ainsi à une exécution incohérente. La figure 5.4 illustre le problème. Sur cet exemple, lorsque le processus p_3 subit une défaillance, il revient vers son dernier point de reprise. Le protocole fait également revenir le processus p_5 pour renvoyer le message m_7 ; les messages m_1 et m_3 sont enregistrés. Le message m_8 aurait été le prochain message reçu par p_3 s'il n'avait pas subi de défaillance. Lorsque le recouvrement commence, les messages m_1, m_3, m_7 et m_8 peuvent être renvoyés à p_3, alors qu'ils sont causalement dépendants. Au niveau de p_3, il est à priori impossible de trouver un ordre causal entre eux. Il est donc nécessaire d'avoir une information supplémentaire pour gérer les causalités.

Une première solution pour résoudre ce problème est d'enregistrer les déterminants de tous les messages, afin de pouvoir les rejouer dans l'ordre. Cependant, comme décrit dans le Chapitre 2, enregistrer les déterminants est coûteux.

Dans ce protocole, les retours arrière suivent le chemin des dépendances causales étant donné que le récepteur d'un message fait revenir son émetteur. Cependant, les points de reprise et les messages enregistrés provoquent des ruptures dans le chemin afin d'éviter de revenir au tout début de l'exécution. Ainsi, les messages dépendants de messages orphelins peuvent être rejoués durant le recouvrement. Sur la figure 5.4, c'est parce que le message m_3 est enregistré qu'il peut être renvoyé par p_3 avant que le message m_1 ne soit rejoué, même si m_3 dépend du message orphelin m_2.

Afin de gérer ce problème, nous introduisons la notion de *phases d'exécution* : Tous les messages sur un chemin causal qui n'est pas interrompu par un point de reprise ou un message enregistré sont dans la même phase. La phase est incrémentée chaque fois que le chemin est interrompu. Ainsi, chaque processus se voit attribuer une phase initialisée à 1. La phase est attachée à chaque message et la phase d'un message est celle de son émetteur. La phase d'un processus est mise à jour comme suit :

1. Un processus incrémente sa phase lorsqu'il prend un point de reprise
2. Lorsqu'un processus reçoit un message non enregistré, il met à jour sa phase en prenant le maximum entre sa phase et celle du message

3. Lorsqu'un processus reçoit un message enregistré, il fait en sorte que sa phase soit supérieure à celle du message. Il prend donc le maximum entre la phase du message incrémentée de 1 et sa phase.

Pendant le recouvrement, les phases sont utilisées afin d'assurer qu'un message est envoyé seulement si les messages orphelins dont il dépend ont été rejoués. Sur la figure 5.4, étant donné que les phases sont initialisées à 1, la phase du message m_1 est 1. Étant donné que ce message est enregistré, en le recevant, le processus p_3 va comparer sa phase actuelle qui est 2 (car il a pris un point de reprise avant la réception de m_1), à celle de m_1 qui est 1 et va garder sa phase actuelle. Puis il envoie m_2 dont la phase est 2 au processus p_4 qui va mettre sa phase à 2 puis il envoie le message m_3 dont la phase est 2 et qui est enregistré. En le recevant, p_3 incrémente sa phase à 3. Et ainsi de suite. Après la défaillance de p_3, les messages m_0, m_2, m_6 et m_7 seront rejoués selon leur phase.

Cependant, selon la nature du message, les conditions de rejeu ne sont pas les mêmes.

Si le message est rejoué (son émetteur a effectué un retour arrière avant son émission), et s'il y a des orphelins dont il dépend, ces orphelins ont été reçus avant son dernier point de reprise. En effet, étant donné que l'émetteur du message effectue un retour arrière, et que le message reçu est un orphelin, le processus n'est pas revenu avant sa réception. Sur la figure 5.4, m_7 est un message rejoué. Il dépend du message orphelin m_5. Ainsi, la phase du message rejoué est supérieure à celle des messages orphelins étant donné qu'ils sont séparés par un point de reprise. Il peut donc être rejoué une fois que ces messages, dont la phase est strictement inférieure à la sienne sont réémis.

Si le message est un message enregistré, les messages orphelins dont il dépend peuvent être dans la même phase (étant donné que c'est le récepteur du message enregistré qui incrémente sa phase et non son émetteur). Sur la figure 5.4, le message m_3 est un message enregistré qui dépend du message orphelin m_2 qui a la même phase que lui. Ainsi, pour pouvoir envoyer un message enregistré, il faut attendre que tous les orphelins dont il dépend soient réémis, *i.e.*, tous les messages orphelins dont la phase est inférieure ou égale à la sienne.

Pour envoyer des messages dont les émetteurs ne sont pas revenus en arrière (par exemple m_8 sur la figure 5.4), la condition est la même que pour les messages enregistrés étant donné qu'ils peuvent dépendre de messages orphelins à la même phase. Sur la figure 5.4, le message m_8 dépend du message orphelin m_4 et les phases des deux messages sont égales à 3.

Pour résumer, supposons qu'un message m, dont la phase est ρ doit être envoyé :

1. Si m est un message rejoué, il ne sera envoyé que lorsque tous les messages orphelins dont la phase est ρ' telle que $\rho' < \rho$ sont envoyés.
2. Si m est un message enregistré, il ne sera renvoyé que lorsque tous les messages orphelins dont la phase est ρ' telle $\rho' \leq \rho$ sont renvoyés.
3. Si m est un message envoyé par un processus qui n'a pas fait de retour arrière, il ne sera envoyé que lorsque tous les messages à une phase ρ' telle que $\rho' \leq \rho$ sont renvoyés.

Sur la figure 5.4, après la défaillance et le redémarrage du processus p_3, étant donné que la phase du message m_1 est 1 et qu'il n'y a pas de messages orphelins dans la phase 1, c'est le seul message qui peut être envoyé. Une fois que le processus p_3 le reçoit, il peut envoyer le message orphelin m_2 puis m_3 peut être envoyé. À la réceptions de m_3, le processus p_3 peut envoyer m_4 puis m_5. Une fois que ces deux messages ont été envoyés, m_7 et m_8 peuvent être envoyés. L'ordre causal est ainsi respecté.

Afin de gérer les phases, après une défaillance, un processus appelé *processus de redémarrage* est lancé. Son rôle est d'assurer que tous les orphelins d'une phase sont reçus avant que les processus n'envoient leurs messages, conformément aux règles décrites ci-dessus. Les algorithmes des processus de l'application et du processus de redémarrage sont détaillés dans le paragraphe 5.3.4.

5.3.4 Pseudo code

La figure 5.5 représente le pseudo code du protocole en fonctionnement sans faute. La figure 5.6 présente l'algorithme des processus de l'application lorsqu'une défaillance se produit.

Durant l'exécution sans fautes, chaque message doit être acquitté par son récepteur (ligne 20 de la figure 5.5) afin que l'émetteur sache si le message doit être enregistré ou pas (lignes 24-25 de la figure 5.5). La phase du processus est incrémentée à chaque point de reprise (ligne 31 de la figure 5.5) et mise à jour chaque fois qu'un message est reçu (lignes 15, 17 de la figure 5.5). La date du processus est incrémentée chaque fois qu'il envoie ou reçoit un message (lignes 10, 18 de la figure 5.5).

Afin de gérer les défaillances, chaque processus sauvegarde, par canal de communication, certaines informations durant l'exécution sans fautes. Pour chaque canal (p_i, p_j) :

- Une structure appelée SPE est utilisée pour sauvegarder, au niveau de p_i, l'époque de réception du dernier message non enregistré envoyé par p_i à p_j pour chaque époque de p_i. Cette information est utilisée pour savoir à quelle époque p_i doit revenir si p_j fait un retour arrière (lignes 13-14 de la figure 5.7).
- Une structure appelée RPP est utilisée afin de sauvegarder la date d'émission du dernier message reçu par p_j pour chaque phase de p_i. Cette information est utilisée pour trouver les messages orphelins entre p_i et p_j si p_i effectue une retour arrière.

Durant le recouvrement, un processus appelé *processus de redémarrage* est lancé afin de calculer la ligne de recouvrement. Son pseudo code est représenté dans la figure 5.7. Lorsqu'un processus effectue un retour arrière, il revient à son dernier point de reprise et diffuse l'époque à partir de laquelle il redémarre à tous les autres processus (lignes 6-12 de la figure 5.6). Lorsqu'un processus reçoit une notification de défaillance, il suspend son exécution et envoie sa table SPE au processus de redémarrage (lignes 13-16 de la figure 5.6). Une fois que le processus de redémarrage a reçu toutes les tables SPE, il commence le calcul de la ligne de recouvrement. Le calcul se termine lorsque plus aucun message n'est vu comme envoyé mais pas reçu (lignes 9-15 de la figure 5.7).

La ligne de recouvrement, contenant l'époque à laquelle chaque processus doit démarrer et la date du processus au début de cette époque, est envoyée à tous les processus

Variables Locales:
1: $Statut_i \leftarrow Actif$ /* Statut du processus, $Actif$, $Bloque$ or $Redemarre$ */
2: $Date_i \leftarrow 1$; $Epoque_i \leftarrow 1$; $Phase_i \leftarrow 1$ /* Date de l'évènement courant, époque et phase du processus P_i */
3: $min_epoque \leftarrow 1$ /* Plus petite époque du système, utilisée pour supprimer les données obsolètes */
4: $NonAck_i \leftarrow \emptyset$ /* Liste des messages émis par le processus P_i et pas encore acquittés */
5: $Enregistres_i \leftarrow \emptyset$ /* Liste des messages enregistrés par P_i */
6: $SPE_i \leftarrow [\perp, ..., \perp]$ /* Informations sur les messages émis par époque. $SPE_i[Epoque_{emission}].date$ est la date de P_i au début de l'époque $Epoque_{emission}$ */
7: $RPP_i \leftarrow [\perp, ..., \perp]$ /* Informations sur les messages reçus par phase */
8: $RA_i \leftarrow [\perp, ..., \perp]$ /* $RA_i[j].epoch$ est l'époque dans laquelle P_j doit redémarrer après une défaillance ; $RA_i[j].date$ est la date correspondante */
9: **À l'envoi du message** msg **au processus** P_j
10: $Date_i \leftarrow Date_i + 1$
11: $NonAck_i \leftarrow NonAck_i \cup$ (P_j, $Epoque_i$, $Date_i$,msg)
12: Envoyer (msg, $Date_i$, $Epoque_i$, $Phase_i$) au processus P_j

13: **À la réception de (**msg**,** $Date_{emission}$**,** $Epoque_{emission}$**,** $Phase_{emission}$**) du processus** P_j
14: **si** $Epoque_{emission} < Epoque_i$ **alors** // Enregistre
15: $Phase_i \leftarrow Max(Phase_i, Phase_{emission} + 1)$
16: **sinon**
17: $Phase_i \leftarrow Max(Phase_i, Phase_{emission})$
18: $Date_i \leftarrow Date_i + 1$
19: $RPP_i[Phase_i][j].date \leftarrow Date_{emission}$
20: Envoyer (***Ack***, $Epoque_i$, $Date_{emission}$) au processus P_j
21: Delivrer msg à l'application

22: **À la réception de (*Ack*,** $Epoque_{recu}$**,** $Date_{emission}$**) du processus** P_j
23: Supprimer (P_j, $Epoque_{emission}$, $Date_{emission}$, msg) de $NonAck_i$
24: **si** $Epoque_{emission} < Epoque_{recu}$ **alors** // Enregistrer
25: $Enregistres_i \leftarrow Enregistres_i \cup$ (P_j, $Epoque_{emission}$, $Date_{emission}$, $Phase_{emission}$, $Epoque_{recu}$, msg)
26: **sinon**
27: $SPE_i[Epoque_{emission}][P_j].epoch_{recu} \leftarrow Epoque_{recu}$

28: **Au point de reprise**
29: Sauvegarder ($Epoque_i$, $ImagePs_i$, RPP_i, SPE_i, $Enregistres_i$, $Phase_i$, $Date_i$) sur support stable

30: $Epoque_i \leftarrow Epoque_i + 1$
31: $Phase_i \leftarrow Phase_i + 1$
32: $SPE_i[Epoque_i].date \leftarrow Date_i$

33: **À la suppression des données obsolètes**
34: Envoyer(***Suppression***, $Epoque_i$) à tous les processus de l'application
35: $min_epoque \leftarrow Epoque_i$

36: **À la réception de (*Suppression*,** $Epoque_{supp}$**) du processus** P_j
37: $min_epoque \leftarrow Min(min_epoque, Epoque_{supp})$
38: **si** Message reçu de tous les processus de l'application **alors** // Suppression des données
39: **pour tout** ($epoque$, $ImagePs$, RPP, SPE, $Phase$, $Date$) en Support stable **telle que** $epoque < min_epoque$ **faire** // Suppression des données
40: Supprimer ($epoque$, $ImagePs$, RPP, SPE, $Phase$, $Date$) du support stable
41: **pour tout** (P_j, $Epoque_{emission}$, $Date_{emission}$, $Phase_{emission}$, $Epoque_{recu}$, msg) $\in Enregistres_i$ en Support Stable **telle que** $Epoque_{recu} < min_epoque$ **faire** // Suppression des messages
42: Supprimer (P_j, $Epoque_{emission}$, $Date_{emission}$, $Phase_{emission}$, $Epoque_{recu}$, msg) de $Enregistres_i$

FIGURE 5.5 – Algorithme du protocole pour les processus de l'application

Variables Locales:
1: $Statut_i$, $Date_i$; $Epoque_i \leftarrow 1$; $Phase_i$, $Enregistres_i$, SPE_i, RPP_i
2: $OrphCompte_i \leftarrow [\perp, ..., \perp]$ /* $OrphCompte_i[phase]$ est le nombre de processus qui envoie un message orphelin au processus P_i à la phase *phase* */
3: $OrphPhases_i \leftarrow \emptyset$ /* Phases dans lesquelles le processus P_i a reçu un message orphelin */
4: $EnrPhases_i \leftarrow \emptyset$ /* Phases des messages enregistrés que le processus P_i doit renvoyer */
5: $RejeuEnr_i \leftarrow [\perp, ..., \perp]$ /* $RejeuEnr_i[phase]$ est la liste des messages enregistrés par P_i qui doivent être rejoués dans la phase *phase* */

6: **À la défaillance du processus** P_i
7: Récupérer la dernière ($Epoque_i$, $ImagePs_i$, RPP_i, SPE_i, $Enregistres_i$, $Phase_i$, $Date_i$) du support stable
8: Redémarrer de $ImagePs_i$
9: $Statut_i \leftarrow Redemarre$
10: Envoyer (***Redemarrage***, $Epoque_i$, $Date_i$) à tous les processus et au processus de redémarrage
11: Envoyer SPE_i au processus de redémarrage
12: **Attendre jusqu'à** réception de ***NotifRedem*** du processus de redémarrage

13: **À la réceptin de (*Redemarrage*, $Epoque_{re}$, $Date_{re}$) du processus** P_j
14: $Statut_i \leftarrow Bloque$
15: Envoyer SPE_i au processus de redémarrage
16: **Attendre jusqu'à** réception de ***NotifRedem*** du processus de redémarrage

17: **À la réception de (*NotifRedem*, RA_{redem}) du processus de redémarrage**
18: **si** $RA_{redem}[i].epoch < Epoque_i$ **alors** // Il faut redémarraer
19: Récupérer ($RA_{redem}[i].epoch$, $ImagePs_i$, RPP_i, SPP_i, $Enregistres_i$, $Phase_i$, $Date_i$) du support stable
20: $Statut_i \leftarrow Redemarre$
21: **pour tout** *phase* **telle que** $RPP_i[phase][j].date > RA_{redem}[j].date$ **faire** // Recherche des phases avec des orphelins
22: $OrphPhases_i \leftarrow OrphPhases_i \cup phase$
23: $OrphCompte_i[phase] \leftarrow OrphCompte_i[phase] + 1$
24: **pour tout** (P_j, $Epoque_{emission}$, $Date_{emission}$, $Phase_{emission}$, $Epoque_{recu}$, *msg*) $\in Enregistres_i$ **tel que** $Epoque_{recv} \geq RA_{redem}[j].epoque$ **faire** // Recherche des message enregistrés à renvoyer
25: $EnrPhases_i \leftarrow EnrPhases_i \cup Phase_{emission}$
26: $RejeuEnr_i[Phase_{emission}] \leftarrow RejeuEnr_i[Phase_{emission}] \cup$ (P_j, $Epoque_{emission}$, $Date_{emission}$, $Phase_{emission}$, $Epoque_{recv}$, *msg*)
27: Envoyer (***Orphan***, $Statut_i$, $Phase_i$, $OrphPhases_i$, $EnrPhases_i$) au processus de redémarrage

28: **À l'envoi du message** *msg* **au processus** P_j
29: **attendre jusqu'à** $Statut_i = Actif$
30: Utiliser la fonction de l'Algorithme A

31: **À la réception de (*msg*, $Date_{emission}$, $Epoque_{emission}$, $Phase_{emission}$) du processus** P_j
32: **si** $Date_{emission} > RPP_i[Phase_i][j].date$ **alors** // *msg* est reçu pour la première fois
33: Utiliser la fonction de l'Algorithme 1
34: **sinon si** $\exists phase$ **telle que** $Date_{emission} = RPP_i[phase][j].date$ **alors** // Dernier orphelin de P_j dans *phase*
35: $OrphCompte_i[phase] \leftarrow OrphCompte_i[phase] - 1$
36: **si** $OrphCompte_i[phase] = 0$ **alors** // Message orphelin dans cette phase
37: Envoyer (***AuncunOrphelinPhase***, *phase*) processus de redémarrage

38: **À la réception de (*PhasePrete*, *Phase*) du processus de redémarrage**
39: **si** $RejeuEnr_i[Phase] \neq \emptyset$ **alors** // Envoyer les messages enregistrés s'il y en a
40: Envoyer $msgs \in RejeuEnr_i[Phase]$
41: **si** $(Statut_i = Redemarre \wedge Phase_i = Phase + 1) \vee (Statut_i = Bloque \wedge Phase_i = Phase)$ **alors**
42: $Statut_i = Actif$

43: **À la réception de (*Ack*, $Epoque_{recu}$, $Date_{emission}$) du processus** P_j
44: Utiliser la fonction de l'Algorithme 1

45: **Au point de reprise**
46: Utiliser la fonction de l'Algorithme 1

47: **À la suppression des données obsolètes**
48: Utiliser les fonctions de l'Algorithme 1

FIGURE 5.6 – Algorithme du protocole pour les processus de l'application après une défaillance

Variables Locales:

1: $TableDependance \leftarrow [\perp, ..., \perp]$ /* $TableDependance[j][Epoque_{send}][k].epoch_{recv}$ est SPE du processus P_j */
2: $RedemarrePhase \leftarrow \emptyset$ /* $RedemarrePhase[phase]$ contient la liste des processus ayant redémarré à la phase $phase$ */
3: $BloquePhase \leftarrow \emptyset$ /* $BloquePhase[phase]$ contient la liste de processus qui n'ont pas redémarré et des messages enregistrés bloqués dans la phase $phase$ */
4: $NbOrphelinPhase \leftarrow \emptyset$ /* $NbOrphelinPhase[phase]$ est le nombre de processus attendant au moins un orphelin dans la phase $phase$ */

5: **À la réception de (*Redemarrage*, $Epoque_{re}$, $Date_{re}$) du processus P_j**
6: $RA_{redem}[j].epoch \leftarrow Epoque_{re}$
7: $RA_{redem}[j].date \leftarrow Date_{re}$
8: **attendre jusqu'à** $TableDependance$ est complète
9: $RA_{tmp} \leftarrow [\perp, ..., \perp]$
10: **repeter**
11: **pour tout** P_j **tel que** $RA_{tmp}[j] \neq RA_{redem}[j]$ **faire**
12: $RA_{tmp}[j] \leftarrow RA_{redem}[j]$
13: **pour tout** P_k **tel que** $TableDependance_i[k][Epoque_{emission}][j].Epoque_{recu} \geq RA_{redem}[j].epoch$ **faire**
14: $RA_{redem}[k].epoque \leftarrow min(RA_{redem}[k].epoque, Epoque_{emission})$ /* Si $Epoque_{emission}$ est choisie, $RA_{redem}[k].date$ est mise à jour */
15: **jusqu'à** $RA_{redem} = RA_{tmp}$
16: Envoyer (***NotifRedem***, RA_{redem}) à tous les processus de l'application

17: **À la réception de** SPE_j **du processus** P_j
18: $TableDependance[j] \leftarrow SPE_j$

19: **À la réception de (*OrphelinNotification*, $Status_j$,$Phase_j$,$OrphPhases_j$, $EnrPhases_j$) from** P_j
20: **si** $Status_j = Rolled - back$ **alors**
21: $RedemarrePhase[Phase_j] \leftarrow RedemarrePhase[Phase_j] \cup P_j$
22: **sinon**
23: $BloquePhase[Phase_j] \leftarrow BloquePhase[Phase_j] \cup P_j$
24: **pour tout** $phase \in EnrPhases_j$ **faire**
25: $BloquePhase[phase] \leftarrow BloquePhase[phase] \cup P_j$
26: **pour tout** $phase \in OrphPhases_j$ **faire**
27: $NbOrphelinPhase[phase] \leftarrow NbOrphelinPhase[phase] + 1$
28: **si *OrphelinNotification*** a été reçue de tous les processus de l'applications **alors**
29: Démarrer **NotifierPhases**

30: **À la réception de (*AucunOrphlinPhase*, $Phase$) du procesus P_j**
31: $NbOrphelinPhase[Phase] \leftarrow NbOrphelinPhase[Phase] - 1$
32: Démarrer **NotifierPhases**

33: **NotifierPhases**
34: **pour tout** $phase$ **telle que** $\nexists phase' \leq phase \wedge NbOrphelinPhase[phase'] > 0$ **faire** // Notification pour les phases qui ne dépendent d'aucun orphelin
35: Envoyer (***PhasePrete***, $phase$) à tous les processus dans $BloquePhase[phase]$
36: Envoyer (***PhasePrete***, $phase$) à tous les processus dans $RedemarrePhase[phase + 1]$

FIGURE 5.7 – Algorithme du procotole pour le processus de redémarrage

de l'application (ligne 16 de la figure 5.7). Lorsqu'un processus reçoit la ligne de recouvrement du processus de redémarrage, il calcule la liste des phases pour lesquelles il attend un message orphelin en utilisant la structure RPP (lignes 21-23 de la figure 5.6). Puis, il calcule la liste des messages enregistrés qu'il doit envoyer (lignes 24-26 de la figure 5.6). Enfin, il envoie au processus de redémarrage sa phase actuelle, son statut (retour arrière ou pas), les phases auxquelles il attend un message orphelin et les phases des messages enregistrés qu'il doit envoyer (ligne 27 de la figure 5.6).

Une fois que le processus de redémarrage a reçu toutes ces informations, il commence à notifier les processus. Pour ce faire, lorsqu'un processus reçoit l'ensemble des messages orphelins qu'il attend à une phase, il notifie le processus de redémarrage (lignes 34-37 de la figure 5.5). Lorsque tous les orphelins d'une phase inférieure ou égale ρ ont été reçus, le processus de redémarrage notifie tous les processus qui n'ont pas effectué de retour arrière et dont la phase est inférieure ou égale à ρ ainsi que tous les messages enregistrés bloqués à une phase inférieure ou égale à ρ. Il notifie également les processus ayant effectué un retour arrière qui sont à une phase inférieure à ρ (lignes 33-36 de la figure 5.7).

5.3.5 Suppression des données obsolètes

Si E est la plus grande époque du système commune à tous les processus, après une défaillance, les processus n'effectueront pas de retour arrière vers une époque inférieure à E car les messages émis à une époque plus petite que E sont enregistrés. Sur la figure 5.3, la plus petite époque commune à tous les processus est l'époque 3. Après une défaillance, étant donné que le message m_7 est enregistré, les processus ne redémarrent pas à partir d'une époque antérieure à 3.

Il n'est donc pas nécessaire de garder les points de reprise dont l'époque est inférieure à E ni les messages enregistrés qui sont reçu dans une époque inférieure à E.

Pour ce faire, périodiquement, les processus s'échangent leur époque actuelle (ligne 34 de la figure 5.5). Une fois qu'un processus a reçu les époques de tous les autres processus, il supprime du support stable l'ensemble des points de reprise qui ont une époque inférieure à l'époque minimum (lignes 39-40 de la figure 5.5) ainsi que tous les messages qui ont été reçus à une époque inférieure à celle-ci (ligne 41-42 de la figure 5.5) (ces messages peuvent être sur support stable ou en mémoire).

5.3.6 Preuve

Dans cette section, nous commençons par donner quelques définitions utilisées dans la preuve avant de prouver qu'après une défaillance, l'exécution reste correcte.

Définition 5.1 *Sous l'hypothèse du déterminisme des émissions, une exécution correcte est une exécution telle que :*
- *Chaque processus envoie une séquence correcte de messages.*
- *L'ordre causal entre les réceptions des messages est respecté.*

Afin de prouver cela, nous considérons l'état à partir duquel une application redémarre après une défaillance, *i.e*, la ligne de recouvrement calculée par la figure 5.7.

Chaque processus p sauvegarde un ensemble de points de reprise H_p^i où i représente le numéro du point de reprise.

Définition 5.2 (Intervalle de points de reprise et époque) *Un intervalle de points de reprise I_p^i représente l'ensemble des évènements entre H_p^i et H_p^{i+1}. i représente l'époque du processus.*

Soit un processus p qui fait un retour arrière vers le point de reprise H_p^x après une défaillance.

Définition 5.3 (Messages annulés) *Soit $\mathscr{R}$ l'ensemble des messages annulés d'un processus p.*

$m \in \mathscr{R}$ si et seulement si $reception(m) \in V_p$ et $reception(m) \in I_p^y$ avec $y \geq x$.

$\mathscr{R}|(q,p)$ représente le sous-ensemble des messages de $\mathscr{R}$ reçus sur le canal (q,p).

Définition 5.4 (Messages enregistrés) *Soit $\mathscr{L}$ l'ensemble des messages enregistrés par processus q.*

$m \in \mathscr{L}$ si et seulement si, pour un processus p, $envoi(m) \in I_q^z$ et $reception(m) \in I_p^y$ avec $y \geq z$.

$\mathscr{L}|(q,p)$ représente le sous-ensemble des message dans $\mathscr{L}$ envoyés sur le canal (q,p).

Définition 5.5 (Messages rejoués) *Soit $\mathscr{S}$ l'ensemble des message rejoués par un processus p.*

$m \in \mathscr{S}$ si et seulement si $envoi(m) \in V_p$ et $envoi(m) \in I_p^y$ avec $y \geq x$.

$\mathscr{S}|(p,q)$ représente le sous-ensemble des messages dans $\mathscr{S}$ envoyés sur le canal (p,q). Noter que $\mathscr{L} \cap \mathscr{S} = \emptyset$

La date courante d'un processus p est notée $date_p(p)$. La date d'un évènement $e \in V_p$, notée $date(e)$ est la date $date_p(p)$ lorsque e se produit.

Définition 5.6 (Messages orphelins) *Soit $\mathscr{O}$ l'ensemble des messages orphelins et m un message sur le canal (p,q).*

$m \in \mathscr{O}$ si et seulement si $date(envoi(m)) > date_p(p)$ et $date(reception(m)) < date_p(q)$.

$\mathscr{O}|(p,q)$ représente le sous-ensemble des messages de $\mathscr{O}$ reçus sur le canal (p,q).

Noter que $\mathscr{O} \cap \mathscr{R} = \emptyset$ et $\mathscr{O} \subset \mathscr{S}$.

Définition 5.7 (L'ensemble des évènements dans une phase) *Soit $Ph(p)$ la phase du processus p.*

Soit V_p^i l'ensemble des évènements d'un processus p à la phase i. Un évènement $e \in V_p^i$ si et seulement si $Ph(p) = i$ lorsque e se produit.

Définition 5.8 (Phase d'un message) *La phase d'un message $Ph_m(m) = k$ avec $envoi(m) \in V_p^k$*

Définition 5.9 (Ensemble des messages orphelins dans une phase) *Soit $\mathscr{O}_{=k}$ l'ensemble des messages orphelins à une phase k.*

Un message $m \in \mathscr{O}_{=k}$ si et seulement si $Ph_m(m) = k$.

De la même manière, nous définissons $\mathscr{O}_{\leq k}$ comme étant l'ensemble des messages orphelins dans une phase inférieure ou égale à k.

Définition 5.10 (Ensemble des messages annulés dans une phase) *Soit $\mathscr{R}_{=k}$ l'ensemble des messages annulés dans la phase k.*

$m \in \mathscr{R}_{=k}$ si et seulement si $m \in \mathscr{R}$ et $reception(m) \in V_p^k$

Notons que l'ensemble des messages orphelins est construit selon la phase des émetteurs alors que l'ensemble des messages annulés est construit selon la phase des récepteurs.

Afin de prouver qu'une séquence de messages valide est envoyée par chaque processus, nous commençons par prouver qu'il n'y a aucun message perdu. Puis nous prouvons que tous les messages annulés seront renvoyés.

Lemme 5.1 *Soient deux processus p et q avec p qui envoie un message sur le canal (p, q).*

Si q fait un retour arrière, $\mathscr{R}|(p, q) \setminus \mathscr{S}|(p, q) \in \mathscr{L}|(p, q)$.

Preuve Soient $m \in \mathscr{R}|(p, q)$ et H_q^i le point point de reprise vers lequel q fait son retour arrière.

Comme $m \in \mathscr{R}|(p, q)$ alors $reception(m) \in I_q^k$ avec $k \geq i$.

Si $m \notin \mathscr{S}|(p, q)$ alors $envoi(m) \in I_p^j$ avec $j < k$. Donc $m \in \mathscr{L}$ par définition d'un message enregistré.

Voici quelques caractéristiques de l'algorithme :

Proposition 1 *Soient m_i et m_j deux messages. Si $m_i \rightarrow m_j$ alors $Ph_m(m_i) \leq Ph_m(m_j)$ (lignes 14-17 et 31 de la figure 5.5).*

Proposition 2 (Condition de renvoi d'un message à rejouer) *Soit $m_j \in \mathscr{S}$. m_j est renvoyés si et seulement si $\mathscr{O}_{<Ph_m(m_j)} = \emptyset$ (ligne 35 de la figure 5.7).*

Proposition 3 (Condition de renvoi d'un message non rejoué) *Soit $m_j \notin \mathscr{S}$. m_j est renvoyé si et seulement si $\mathscr{O}_{\leq Ph_m(m_j)} = \emptyset$ (ligne 36 de la figure 5.7).*

Nous donnons maintenant le condition de renvoi un message annulé dont la phase est x.

Lemme 5.2 *$\forall m_i \in \mathscr{R}_{=x}$, m_i est renvoyé si et seulement si $\mathscr{O}_{<x} = \emptyset$.*

Preuve Comme $m_i \in \mathscr{R}$ alors $m_i \in \mathscr{S}$ ou $m_i \in \mathscr{L}$ selon le lemme 5.1.

1. Si $m_i \in \mathscr{S}$, $Ph_m(m_i) \leq x$ (ligne 17 de la figure 5.5), m_i est envoyé si $\mathscr{O}_{<x} = \emptyset$ selon la proposition 2.

2. Si $m_i \in \mathscr{L}$: m_i est envoyé si $\mathscr{O}_{\leq Ph_m(m_i)} = \emptyset$ selon la proposition 3. Étant donné que $m_i \in \mathscr{L}$ et $m_i \in \mathscr{R}_{=x}$ alors $Ph_m(m_i) < x$ (vue que la phase d'un récepteur est toujours supérieure à celle du message enregistré reçu lignes 14-15 de la figure 5.5). Donc m_i est envoyé si $\mathscr{O}_{<x} = \emptyset$.

Lemme 5.3 *Tous les messages annulés sont renvoyés*

Preuve Soit $\mathscr{O}_{min_phase}$ l'ensemble des messages orphelins de plus petite phase.
Selon le lemme 5.2, $\forall m_i \in \mathscr{R}_{<min_phase}$, m_i est envoyé.
Et $\forall m_j \in \mathscr{O}_{=min_phase}$, m_j est renvoyé selon la proposition 2.
Donc $\mathscr{O}_{min_phase} = \emptyset$ and $\forall m_i \in \mathscr{R}_{<min_phase+1}$, m_i est envoyé.
Et ainsi de suite jusqu'à ce que $\mathscr{O} = \emptyset$.

Nous prouvons maintenant que la causalité des messages reçus est respectée. La causalité est corrompue si il existe deux message m_i et m_j tels que $m_i \to m_j$ mais m_j est reçu avant m_i pendant le redémarrage. Si tous les processus font un retour arrière vers un état qui ne dépend pas de m_i, la condition est trivialement vérifiée vue que m_j ne peut être envoyé avant que m_i n'ait été rejoué.

Nous commençons par prouver que si l'exécution est dans un état où m_i et m_j peuvent tous les deux être émis, alors il y a un message orphelin dans la causalité entre m_i et m_j

Lemme 5.4 *Soient m_i et m_{i+x}, $x \geq 1$ deux messages tels que $m_i \to m_{i+1} \ldots \to m_{i+x}$*
Si m_i et m_{i+x} peuvent tous les deux être émis, alors $\exists m \in \mathscr{O}$ (qui peut être m_i) tel que : $m_i \to \ldots \to m \to \ldots \to m_{i+x}$.

Preuve Étant donné que m_i et m_{i+x} peuvent être émis alors $m_i \in \mathscr{R} \cup \mathscr{O}$.
m_{i+x} peut être renvoyé si $m_{i+(x-1)}$ est reçu. Soit l'émetteur de $m_{i+(x-1)}$ fait un retour arrière et donc $m_{i+(x-1)} \in \mathscr{O}$ ou pas.
Si $m_{i+(x-1)} \notin \mathscr{O}$, $m_{i+(x-2)}$ est reçu. Donc $m_{i+(x-2)}$ est soit un orphelin ou pas. Et ainsi de suite jusqu'à m_{i+1} qui peut être envoyé si m_i est reçu. Étant donné que $m_i \in \mathscr{R} \cup \mathscr{O}$ et que l'émetteur de m_{i+1} ne fait pas de retour arrière (sinon, nous nous serions arrêtés à m_{i+2}), alors $m_i \in \mathscr{O}$.

Considérons la relation entre la phase d'un message orphelin et la phase d'un message rejoué qui dépend de lui

Lemme 5.5 *Soit m_i et m_j deux messages tels que $m_i \to m_j$.*
Si $m_i \in \mathscr{O}$ et $m_j \in \mathscr{S}$ alors $Ph_m(m_i) < Ph_m(m_j)$.

Preuve Étant donné que $m_i \to m_j$ alors selon la Proposition 1, $Ph_m(m_i) \leq Ph_m(m_j)$.
Afin de prouver que $Ph_m(m_i) < Ph_m(m_j)$, nous utilisons la preuve par l'absurde en prouvant que $Ph_m(m_i) = Ph_m(m_j)$ mène à une contradiction.
Supposons que $Ph_m(m_i) = Ph_m(m_j)$. Selon les lignes 14 et 31 de la figure 5.5, il n'y a ni points de reprise ni messages enregistrés sur le chemin des causalités de m_i à m_j (étant donné que ce sont les deux seuls évènements incrémentant une phase). Selon les lignes 9-15 de la figure 5.7, si $m_j \in \mathscr{S}$, $m_i \in \mathscr{R}$. Ceci est impossible car $m_i \in \mathscr{O}$ et $\mathscr{O} \cap \mathscr{R} = \emptyset$.

Enfin, nous prouvons qu'un message dépendant d'un message orphelin ne peut pas être émis avant que le message orphelin ne le soit.

Lemme 5.6 *Si m_i et m_j sont deux messages tels que $m_i \rightarrow m_j$ et $m_i \in \mathscr{O}$, m_j ne peut pas être envoyé avant que m_i ne soit envoyé.*

Preuve Si $m_j \in \mathscr{S}$, m_j en envoyé si et seulement si $\mathscr{O}_{<Ph_m(m_j)} = \emptyset$ (Proposition 2). Si $m_j \notin \mathscr{S}$, m_j est envoyé si et seulement si $\mathscr{O}_{\leq Ph_m(m_j)} = \emptyset$ (Proposition 3). $m_i \in \mathscr{O}_{=Ph_m(m_i)}$. Étant donné que $m_i \rightarrow m_j$, $Ph_m(m_j) \geq Ph_m(m_i)$ selon le Lemme 1.

1. Si $Ph_m(m_i) = Ph_m(m_j)$: alors $\mathscr{O}_{\leq Ph_m(m_j)} \neq \emptyset$. Selon le Lemme 5.5, étant donné que $Ph_m(m_i) = Ph_m(m_j)$, $m_j \notin \mathscr{S}$.
2. Si $Ph_m(m_i) < Ph_m(m_j)$: alors $m \in \mathscr{O}_{<Ph_m(m_j)} \neq \emptyset$.

Théorème 5.1 *Sous l'hypothèse du déterminisme des émissions, après une défaillance, l'exécution est correcte.*

Preuve Les lemmes 5.1 et 6.1 montrent qu'après une défaillance, tous les processus envoient leur séquence valide de messages. Le Lemme 5.6 montre que le protocole force l'ordre causal des réceptions des messages. □

La preuve pour les défaillances multiples est la même car nous n'avons pas précisé que le processus p considéré au début de la preuve redémarre après une défaillance ou après un redémarrage forcé. Lorsqu'un processus non fautif effectue un retour arrière, il ne fait que restaurer son image à partir du point de reprise à partir duquel il doit redémarrer. Un processus fautif a le même comportement. Toutes les informations nécessaires sont de ce fait incluses dans le point de reprise.

5.4 Évaluation

Dans cette section, nous décrivons d'abord le prototype développé dans MPICH2. Puis, nous présentons les évaluations expérimentales. Nous évaluons d'abord les performances en exécution sans faute. Puis, nous évaluons le nombre de processus qui effectuent un retour arrière en cas de défaillance et proposons une solution fondée sur la définition de groupes de processus décrite dans le Chapitre 4, afin de limiter le nombre de processus qui effectuent un retour arrière tout en n'enregistrant qu'un sous-ensemble des messages de l'application.

5.4.1 Description du prototype

Le protocole a été mis en œuvre dans MPICH2 (la même version que celle utilisée dans le Chapitre 4), dans le système de communication Nemesis. Le protocole fonctionne sur TCP, en mémoire partagée et sur Myrinet/MX.

Mise en œuvre des acquittements Nous avons vu dans le paragraphe 5.3.4 que le protocole nécessite que tous les messages soient acquittés avec l'époque de réception afin de contrôler l'enregistrement des messages. Mais si une émission ne peut se terminer tant que l'acquittement n'est pas reçu, cela signifie que les acquittements sont envoyés de manière synchrone, ce qui risque d'avoir un impact sur les performances et plus spécialement la latence des petits messages sur réseaux rapides. Afin de minimiser l'impact sur les performances, le nombre d'acquittements doit être limité. Pour ce faire, nous exploitons la propriété FIFO des canaux. Ceci est illustré sur la figure 5.8. Observons le processus p_1. Au lieu de le forcer à attendre un acquittement pour chaque message, les messages sont copiés en mémoire. Ainsi, une opération d'émission peut se terminer sans attendre l'acquittement. Seuls les messages qui doivent être enregistrés sont explicitement acquittés. Comme les canaux sont FIFO, tous les messages envoyé par p_1 au processus p_2 après le premier message enregistré doivent être enregistrés tant que leur époque d'émission est inférieure à l'époque de réception. Cela reste vrai jusqu'à ce que p_1 change d'époque. Sur la figure 5.8, le message m_4 est le premier message à être enregistré. Une fois que p_1 reçoit l'acquittement pour m_4 qui indique qu'il doit être enregistré, le message m_5 doit automatiquement être enregistré car p_1 n'a pas changé d'époque depuis l'émission de m_4. Ainsi, seul le premier message par époque et par canal qui doit être enregistré, est acquitté. Les messages qui sont automatiquement enregistrés sont marqués comme tels afin d'éviter que p_2 n'envoie un acquittement.

Étant donné que le processus copie en mémoire les messages non acquittés, il faut qu'il puisse les supprimer s'ils ne doivent pas être enregistrés. Pour ce faire, chaque message est identifié par un numéro de séquence d'émission (ssn). Chaque processus attache, sur les messages qu'il envoie, le ssn du dernier message qu'il a reçu du processus destinataire. Lorsqu'un processus reçoit un message avec un ssn attaché, il supprime l'ensemble des messages dont le numéro de séquence d'émission est inférieur au ssn reçu. Sur la figure 5.8, lorsque p_2 envoie le message m_3 au processus p_1, il y attache $ssn = 2$ pour que p_1 sache qu'il peut supprimer les messages m_1 et m_2 de sa mémoire. Si le nombre de messages non acquittés devient trop important, dans le cas où p_2 n'envoie jamais de message à p_1, p_1 peut demander explicitement un acquittement à p_2.

Cette technique est utilisée pour les messages de petite taille. Dans les évaluations présentées dans la suite du paragraphe, la limite de la taille de tels messages est fixée à $1Ko$. Cependant ce paramètre peut être fixé au moment de lancer l'application. Pour les gros messages, une copie supplémentaire de chaque message peut avoir un coût important sur les performances. Ces messages sont donc acquittés à chaque fois sauf s'ils sont marqués comme déjà enregistrés.

Mise en œuvre des données attachées aux messages Les données attachées aux messages sont : l'époque, la phase, le numéro de séquence du message, le numéro de séquence du dernier message reçu (ssn) et un dernier champs qui sert à dire si le processus demande un acquittement pour ce message. La taille de ces données est de 32 octets sur une architecture 64 bits. Les deux derniers champs servent à mettre en œuvre l'acquittement comme décrit dans le paragraphe précédent.

Selon le réseau utilisé, les données ne sont pas attachées de la même manière. Dans

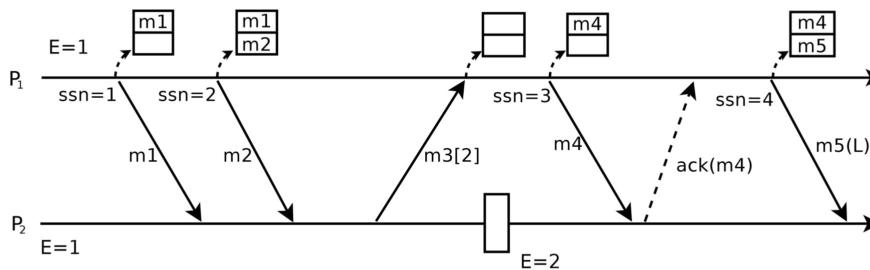


FIGURE 5.8 – Gestion des acquittements pour les messages de petite paille sur un canal de communication

MPICH2, sur TCP et en mémoire partagée, un en-tête est défini pour chaque message contrairement à MX. Les informations sont ajoutées dans l'entête du message. En effet, il est possible d'ajouter autant d'informations que nécessaires dans l'entête.

Sur Myrinet, la fonction *mx_isend* prend un vecteur en entrée auquel il est possible d'ajouter une entrée. Les évaluations présentées étant réalisées sur Myrinet 10G, nous détaillons comment les données du protocole sont envoyées dans ce cas :

- Pour les petits messages (d'une taille inférieure à 1 Ko), les données du protocole sont ajoutées comme un segment supplémentaire à la liste des segments définis pour les données de l'application. Pour de tels messages, envoyer toutes les données dans un même message permet une d'obtenir une meilleure latence qu'en envoyant deux messages séparés.
- Pour les messages de grande taille, les données du protocole sont envoyées dans un message séparé car envoyer un vecteur de segments non contigus dans le même message dans MX peut entraîner une copie supplémentaire des données en mémoire, ce qui aurait un impact sur les performances.

Mise en œuvre de l'enregistrement de messages fondé sur l'émetteur Afin de mettre en œuvre l'enregistrement de messages fondé sur l'émetteur, nous copions les messages dans un tampon pré-alloué en utilisant la fonction *memcpy* de la libc. L'étude présentée dans [74] montre qu'il est théoriquement possible de mettre en œuvre un protocole à enregistrement de messages fondé sur l'émetteur sans introduire de coût supplémentaire car la latence et la bande passante fournies par *memcpy* sont meilleures que celles fournies par Myrinet 10G. Dans notre mise en œuvre, le contenu du message est copié entre l'appel à *mx_isend()* et l'appel à *mx_wait()* correspondant, afin que la copie en mémoire du message et sa transmission sur le réseau puisse se faire en parallèle.

5.4.2 Plate-forme d'expérimentation

Les expériences ont été réalisées sur la plate-forme Grid'5000 [75]. Les évaluations des performances des communications ont été réalisées sur la grappe de Lille sur 2 noeuds équipés de 2 processeurs Intel Xeon E5440 QC (4 cœurs), 8 Go de mémoire et d'une interface Myri-10G 10G-PCIE-8A-C. 45 nœuds avec les même caractéristiques

ont été utilisés pour évaluer les performances des applications. Les expériences évaluant le nombre de messages enregistrés et le nombre de processus à redémarrer après une défaillance ont été exécutées sur la grappe d'Orsay sur 128 nœuds équipés de 2 processeurs AMD Opteron 246, 2 Go de mémoire et d'une interface Gigabit Ethernet. Ces mêmes nœuds ont été utilisés pour extraire les caractéristiques de communications utilisées dans le Chapitre 4. Dans les deux cas, le système d'exploitation est un Linux avec un noyau 2.6.26.

5.4.3 Évaluation des performances en fonctionnement normal

Dans ce paragraphe, nous présentons les résultats des expériences réalisées. Nous commençons par évaluer l'impact de notre protocole sur les performances de l'application Netpipe [76] qui évalue le surcoût sur la latence et la bande passante pour différentes tailles de messages. Par la suite, nous présentons les performances de 6 applications de la suite des *NAS Parallel Benchmark*. Nous n'avons pas inclus la sauvegarde des points de reprise dans l'évaluation car notre objectif ici est d'évaluer le coût de l'enregistrement des messages étant donné la sauvegarde des points de reprise introduit un coût quel que soit le protocole utilisé. Enfin, nous ne disposons pas des ressources nécessaires pour évaluer les avantages d'un protocole de sauvegarde de points de reprise non coordonnés par rapport à un protocole coordonné du point de vue de l'écriture simultanée des points de reprise.

Performance des communications : Afin d'évaluer le surcoût de notre protocole sur les performances des communications, nous avons exécuté un test de type ping-pong en utilisant Netpipe [76]. La figure 5.9 les performances de la version native de MPICH2, les performances de MPICH2 avec le protocole si aucun message n'est enregistré et de l'enregistrement de tous les messages. Évaluer le protocole lorsqu'aucun message n'est enregistré permet d'évaluer la version optimisée de l'acquittement ainsi que l'impact des données attachées.

La figure 5.9 montre que le protocole n'introduit de surcoût que sur les message de petite taille introduisent un surcoût. Afin de mieux voir la dégradation des performances, la figure 5.10 représente les mêmes résultats mais en pourcentage par rapport à la version native de MPICH2. Nous remarquons que pour les messages de taille inférieure à 64 Octets, le surcoût introduit lorsque tous les messages sont enregistrés est d'environ 30% (environ $0.95\mu s$) sur la latence et de 22% sur la bande passante. Ceci est dû au fait que la taille des données attachées aux messages (32 Octets) est supérieure ou égale à la taille du message. Le protocole n'introduit aucun surcoût sur les messages de grande taille (à partir de 1 Ko).

Nous remarquons également qu'il y a un surcoût supplémentaire introduit lorsqu'aucun message n'est enregistré par rapport à l'enregistrement de tous les messages. Ceci s'explique par le fait que le cas où les messages ne sont pas enregistrés est plus complexe du fait de la gestion des acquittements, comme expliqué dans le paragraphe 5.4.1.

Le pic sur la figure 5.10 est dû aux données supplémentaires attachées aux messages. La raison est qu'il y a un plateau dans les performances de la version native de MPICH2

sur MX comme le montre la figure 5.9. Par exemple, dans notre expérience, la latence de MPICH2 est d'environ 3.3μs pour les messages dont la taille varie de 1 à 32 octets, et passe à 4μs après. À cause des données supplémentaires attachées aux données envoyées, le protocole atteint ce plateau avant.

La comparaison des performances avec et sans enregistrement de messages montre que l'enregistrement de messages fondé sur l'émetteur n'a pas d'impact sur les performances lorsque les messages sont enregistrés en mémoire.

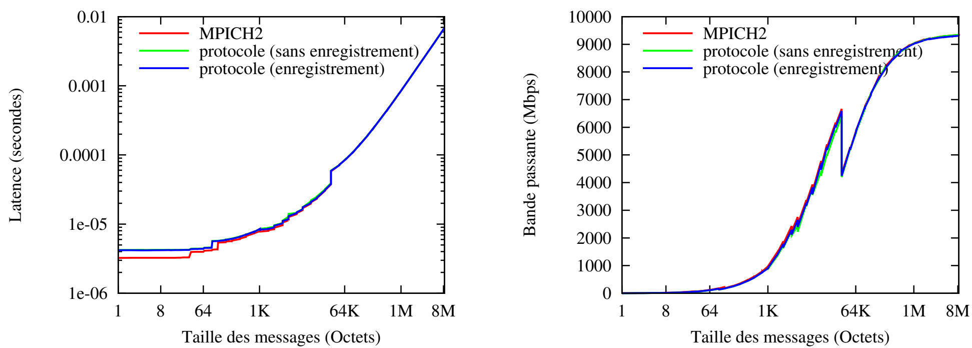


FIGURE 5.9 – Performances de NetPipe sur Myrinet 10G

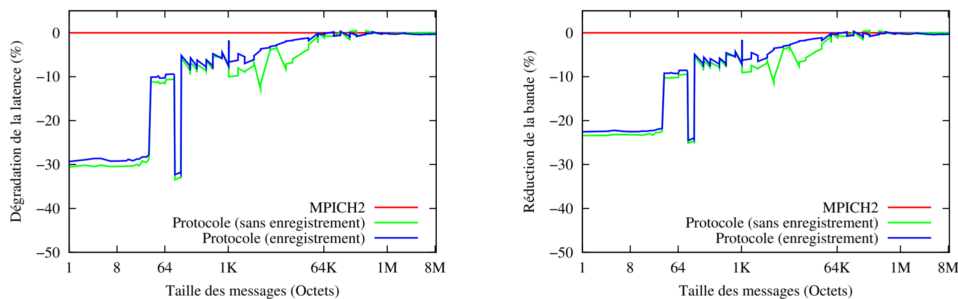


FIGURE 5.10 – Dégradation des performances de NetPipe sur Myrinet 10G

Performances des *NAS Parallel Benchmarks* : Pour nos expériences, nous avons utilisé 6 applications de la suite des *NAS Parallel Benchmarks* [57] sur un problème de taille D sur 256 processus. Pour chaque résultat, chaque application a été exécutée 5 fois et le meilleur temps d'exécution a été choisi.

La figure 5.11 présente une comparaison entre les performances de la version native de MPICH2 aux performances de MPICH2 avec le protocole si aucun message n'est enregistré et si tous les messages sont enregistrés. Les résultats sont normés par rapport au temps d'exécution. La version native de MPICH2 est choisie comme référence.

Les résultats montrent que les performances du protocole lorsqu'aucun message n'est enregistré sont quasiment équivalentes à celles de la version native de MPICH2. De plus, un faible surcoût de moins de 3.5% est introduit lorsque tous les messages sont enregistrés. Nous remarquons que pour LU, les performances du protocole lorsque tous les messages sont enregistrés sont meilleures que les performances si aucun message n'est enregistré. Ceci peut être expliqué par la quantité de mémoire utilisée. En effet, comme expliqué dans le paragraphe 5.4.1, les messages non acquittés sont copiés dans un tampon temporaire, par canal, en attendant la réception de l'acquittement. Cependant, nous continuons à étudier les caractéristiques des applications afin de mieux expliquer ce résultat.

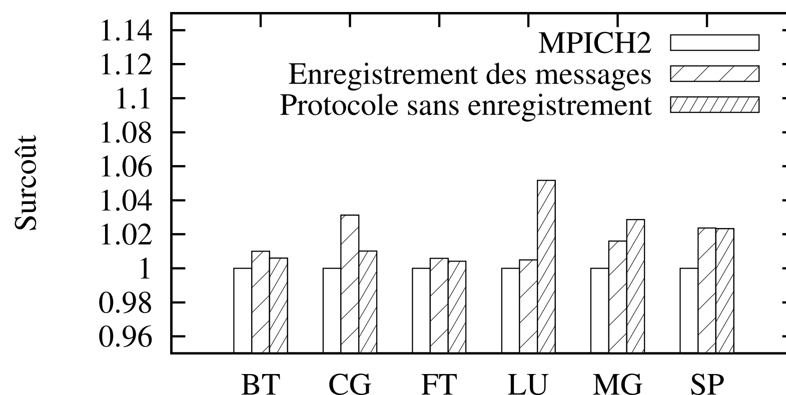


FIGURE 5.11 – Performances des *NAS Benchmarks* sur Myrinet 10G sur 256 processus

5.4.4 Évaluation du nombre de processus à redémarrer :

Afin d'évaluer le nombre de processus à redémarrer après une défaillance, chaque application a été exécutée en enregistrant, sur disque, la table SPE (décrite dans le paragraphe 5.3.4) de chaque processus toutes les 30 secondes. Puis, nous avons analysé les données à posteriori et avons exécuté le protocole de redémarrage : pour chaque version de la table SPE enregistrée, nous avons supposé les défaillances de tous les processus (une seule défaillance à la fois), et pour chaque défaillance, nous avons calculé le nombre de processus à redémarrer.

Les expériences montrent que le redémarrage de tous les processus est nécessaire après une défaillance. Ceci montre que les communications entre les processus créent une dépendance globale entre eux. Ainsi, malgré le fait que théoriquement, le protocole ne force pas de redémarrage global (comme le protocole décrit dans [14]), les applications sont telles qu'en pratique, tous les processus redémarrent après une défaillance. La combinaison de la sauvegarde points de reprises non coordonnés et de l'enregistrement des messages devrait donc prendre en considération les schémas de communications des

applications afin de limiter le nombre de redémarrages tout en garantissant que seul un sous-ensemble des messages de l'application sont enregistrés.

Nous utilisons la caractéristique décrite dans le Chapitre 4 afin de créer des groupes de processus. Dans ce qui suit, nous décrivons l'application du protocole aux groupes ainsi formés.

5.4.5 Utilisation du protocole en définissant des groupes de processus

L'objectif ici est de regrouper les processus qui communiquent fréquemment entre eux. Comme expliqué dans le Chapitre 4, le regroupement des processus vise à limiter le nombre des processus dans les groupes et la taille des messages échangés entre les groupes.

Dans notre protocole, un message est enregistré s'il est envoyé d'une époque vers une époque supérieure. Afin d'adapter le protocole et de l'utiliser sur des groupes de processus, les époques sont attribuées aux groupes, les processus d'un même groupe ont la même époque et les messages inter-groupe sont enregistrés de la même façon que pour les processus : les messages émis des processus du groupe dont l'époque est i vers les processus du groupe d'époque $i+1$, sont enregistrés et les messages reçus par les processus du groupe dont l'époque est i provenant des processus groupe dont l'époque est $i+1$ ne le sont pas. Afin d'assurer que la sauvegarde d'un point de reprise ne fasse pas changer de groupe à un processus, les époques sont espacées de 2. La figure 5.12 montre un exemple de l'utilisation du protocole sur un schéma de communication de l'application *CG* de la suite des *NAS Parallel Benchmarks* de taille C sur 64 processus. Les messages qui vont d'une époque vers une époque supérieure sont enregistrés.

Dans ce qui suit, nous évaluons le nombre de processus à redémarrer et le volume des données enregistrées en utilisant l'algorithme de partitionnement décrit dans le Chapitre 4. Comme expliqué dans le Chapitre 4, l'outil de partitionnement fonctionne pour des applications où la quantité de données échangées entre deux processus est la même dans les deux directions. Nous montrons théoriquement que même pour des applications non symétriques, il est possible de former des groupes de processus minimisant leur taille et le volume des données inter-groupe. Enfin, nous présentons les performances du protocole utilisé sur des groupes de processus définis grâce à l'algorithme de partitionnement.

Évaluation du nombre de processus à redémarrer avec l'outil de partitionnement : Comme expliqué dans le Chapitre 4, l'algorithme de partitionnement crée des groupes de processus de façon à minimiser une fonction de coût donnée. Les paramètres de cette fonction de coût pour ce protocole sont décrits dans le paragraphe 4.4.3.1 du Chapitre 4.

Le tableau 5.1 présente l'évaluation de la quantité de données enregistrées et la taille des groupes obtenus en exécutant l'algorithme présenté dans le Chapitre 4 avec Scotch. La table présente, pour chaque application, le nombre de groupes, la taille minimale et maximale des groupes obtenus, le ratio des processus à redémarrer et le volume des données total et enregistré. Notons que nous supposons que lorsque le processus

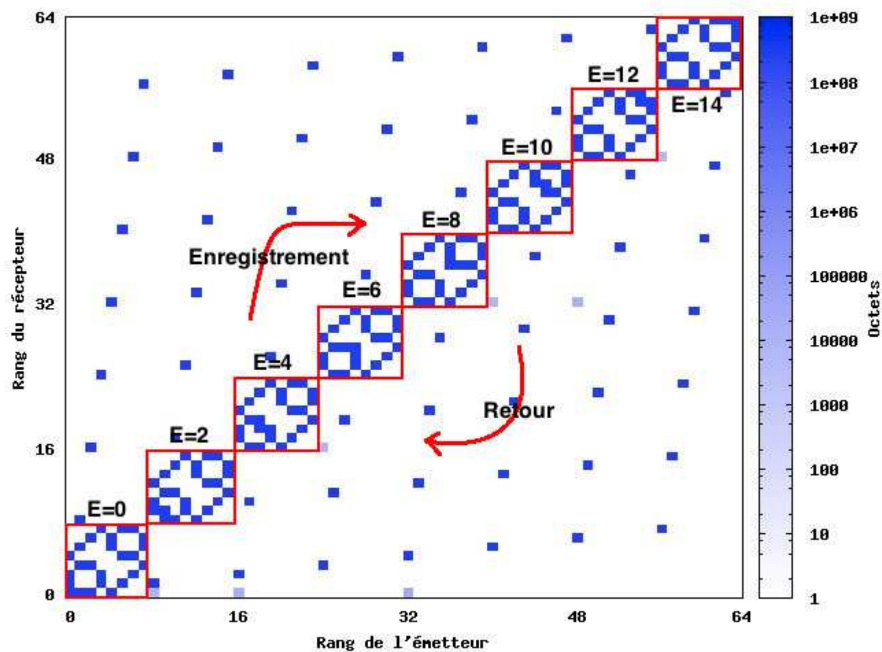


FIGURE 5.12 – Utilisation du protocole en définissant des groupes de processus

d'un groupe subit une défaillance, tous les processus de ce groupe effectuent un retour arrière. L'évaluation a été faite seulement avec des applications possédant des schémas de communications symétriques, c'est-à-dire des applications où le volume des données échangées entre chaque paire de processus est le même dans les deux sens. Nous avons supposé ce type d'application car la sauvegarde des messages n'est pas bidirectionnelle et l'outil de partitionnement ne gère pas ce cas.

Nous observons que pour toutes les applications sauf *FT*, l'outil parvient à trouver un partitionnement tel que le ratio du nombre des processus effectuant un retour arrière est aux environs de 60% en enregistrant moins de 9% des messages. *FT* donne de moins bons résultats pour la quantité de données enregistrées à cause de l'utilisation de la primitive de communication collective *MPI_Alltoall*.

Nous montrons, dans le paragraphe qui suit, qu'il est toujours possible d'avoir moins de 50% de messages enregistrés, et que le nombre de groupe à redémarrer est en moyenne de $(p+1)/2$ (où p est le nombre total de groupes), et ce quelle que soit la nature de l'application, même pour les applications non symétriques.

Calcul théorique du nombre de processus effectuant un retour arrière : Le calcul du nombre de processus effectuant un retour arrière peut être effectué de manière théorique. Supposons que les processus sont regroupés en p groupes numérotés de 1 à p, et que si le processus d'un groupe fait un retour arrière, tous les processus du même groupe effectuent également un retour arrière. Supposons que E_i est l'époque

	Taille	Nb Groupes	Min/Max Taille groupe	Processus à redémarrer	Qte Enr/Totale des données (en GO)
NPB CG	1024	32	32/32	51.56%	455/5606 (8.11%)
NPB FT	1024	2	512/512	75%	216/863 (25%)
NPB LU	1024	16	64/64	53.1%	34/700 (4.8%)
MAESTRO	1024	4	250/262	62.5%	27/309 (8.9%)
PARATEC	1024	16	61/68	53.1%	1285/23914 (5.3%)
LAMMPS	1024	8	127/129	56.2%	0.15/4 (3.8%)

Table 5.1 – Résultats avec l'outil de partitionnement

des processus du groupe i. Tous les messages envoyés des processus des groupes k tels que $E_k < E_i$ sont enregistrés. Ainsi, si le groupe i subit une défaillance, les processus appartenant aux groupes k tels que $E_k < E_i$ n'effectuent pas de retour arrière et les processus appartenant aux groupes j tels que $E_j > E_i$ redémarrent. Ainsi :

- si le groupe avec la plus petite époque E_{min}, effectue un retour arrière, p groupes vont effectuer un retour arrière,
- si le groupe dont l'époque est $E_{min} + q$ ($q > 0$) effectue un retour arrière, tous les groupes tels que $E_i > E_{min} + q$ effectuent un retour arrière, *i.e.*, $(p - q)$ groupes,
- si le groupe avec la plus grande époque effectue un retour arrière, il sera le seul groupe à revenir en arrière.

Considérons p exécutions où une seule défaillance se produit pour chaque exécution, les défaillances étant uniformément distribuées sur les groupes. Sur les p exécutions, le nombre total de groupes effectuant un retour arrière est $p * (p + 1)/2$, ce qui donne $(p + 1)/2$ groupes en moyenne.

Calcul théorique de la quantité de données enregistrée L'ensemble des messages échangés au cours de l'exécution d'une application peut être divisé en 3 ensembles :

- A : Les messages intra-groupe
- B : Les messages inter-groupe enregistrés
- C : Les messages inter-groupe non enregistrés

Si la quantité de données enregistrées (messages de B) est supérieure à 50%, il suffit d'attribuer les époques dans l'ordre inverse. En effet, étant donné que les messages sont enregistrés dans l'ordre croissant des époques, inverser l'ordre des époques reviendrait à enregistrer les messages de C qui engendrera une quantité de données enregistrée inférieure à 50%.

Il est donc possible de créer des groupes de processus de sorte que le volume des messages à enregistrer soit inférieur à 50% et que le nombre de groupes à redémarrer approche 50% en moyenne.

Performances des *NAS Parallel Benchmarks* sur des groupes de processus : Nous avons évalué le protocole appliqué à des groupes de processus en exécution normale sur Myrinet 10G.

La figure 5.13 présente une comparaison entre les performances de la version native de MPICH2, les performances du protocole appliqué à des groupes de processus et du protocole avec enregistrement de tous les messages. Les résultats sont normés par rapport au temps d'exécution. La version native de MPICH2 est choisie comme référence. Les groupes utilisés sont décrits dans la table 5.2. Pour les applications BT, MG et SP, nous avons utilisé la fonction de coût 4.3 étant donné que ces applications ne sont pas symétriques. La table montre que malgré l'utilisation de la fonction de coût 4.3, les groupes créés sont tels que seul un sous ensemble des messages sont enregistrés et que seul un sous ensemble des processus redémarrent en cas de défaillance. Ceci s'explique par le fait que même si ces applications ne sont pas symétriques en taille de messages, elles le sont en nombre de messages. Ainsi, même si le regroupement obtenu n'est pas optimal, il permet de réduire le volume des données.

	Nb Groupes	Min/Max Taille groupe	Processus à redémarrer	Qte Enr/Totale des données (en GO)
NPB BT	5	32/64	52.08%	74/791 (9.38%)
NPB CG	16	16/16	53.12%	220/2318 (9.49%)
NPB FT	2	128/128	75%	432/860 (50%)
NPB LU	8	32/32	56.25%	22/337 (6.63%)
NPB MG	4	64/64	62.5%	6.5/66 (9.85%)
NPB SP	6	32/64	52.5%	140/1446 (9.67%)

TABLE 5.2 – Regroupement de 256 processus

Les résultats montrent que le protocole introduit un faible surcoût et donne de meilleures performances que l'enregistrement de tous les messages pour la plupart des applications. Enregistrer un sous ensemble des messages permet donc d'améliorer les performances.

Nous remarquons aussi que les performances avec les groupes sont meilleures que les performances sans enregistrement de messages présentées sur la figure 5.11 (la différence est néanmoins faible). Afin d'expliquer cette différence, prenons exemple sur la figure 5.8. Sur cette figure, tant que le processus p_1 n'a pas reçu d'acquittement du processus p_2, il sauvegarde les messages en mémoire. Une fois que l'acquittement est reçu (dans le message m_3), p_1 parcourt la liste des messages qu'il a sauvegardé sur le canal afin de savoir quels sont ceux qu'il doit enregistrer et ceux qu'il peut supprimer. Lorsque nous évaluons les performances du protocole sans enregistrement de message, les processus effectuent toutes ces opérations. Une fois qu'un processus reçoit un acquittement pour sauvegarder un message sur un canal, il sait que tous les messages qu'il enverra sur ce canal seront enregistrés tant qu'il ne change pas d'époque. Il n'a donc plus besoin d'effectuer les opérations relatives aux acquittements. C'est ce qu'il se passe lors de l'utilisation des groupes.

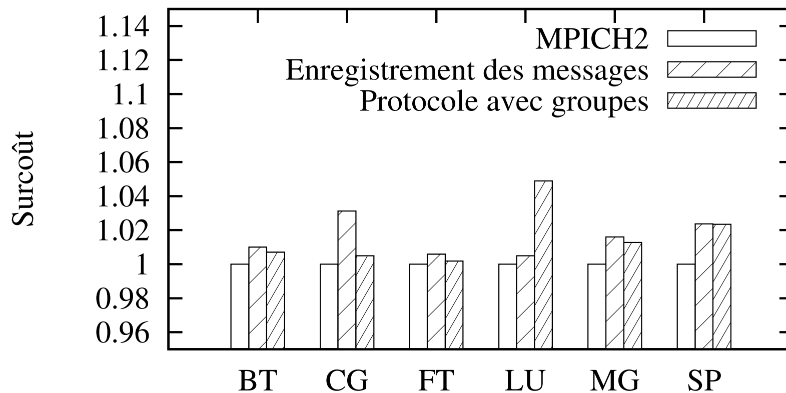


FIGURE 5.13 – Performances du protocole sur des groupes de processus

5.5 Conclusion

Dans ce chapitre, nous avons présenté un protocole de sauvegarde de points de reprise non coordonnés pour les applications à émissions déterministes. Grâce au déterminisme des émissions, les récepteurs des messages orphelins ne font pas de retour arrière évitant ainsi l'effet domino contrairement au protocole de sauvegarde de points de reprise non coordonnés. De plus, les messages n'ont pas besoin d'être reçus dans le même ordre d'une exécution correcte à une autre. De ce fait, les déterminants des messages n'ont pas besoin d'être enregistrés, contrairement aux protocoles à enregistrement de messages.

Lorsqu'une défaillance se produit, le processus fautif redémarre de son dernier point de reprise. Les messages dont il a besoin sont rejoués en redémarrant leurs émetteurs. Afin d'éviter le redémarrage de tous les processus, un sous ensemble des messages échangés sont enregistrés dans la mémoire des émetteurs. Les dates d'émission de ces messages sont également sauvegardées sur la mémoire de l'émetteur.

Nous avons présenté le protocole et ses différents algorithmes. Nous avons également prouvé qu'il garantit une exécution correcte en dépit des défaillances.

Ce protocole a été mis en œuvre dans la bibliothèque MPICH2 sur TCP, en mémoire partagée et sur réseau rapide. Les évaluations sur réseau haute performance montrent que le protocole a un faible impact sur la latence et la bande passante pour les messages de petite taille. Pour les grands messages, le protocole n'introduit aucun surcoût. De plus, l'étude des performances sur les *NAS Parallel Benchmarks* montre un faible impact du protocole comparé à la version native de MPICH2.

L'évaluation du nombre de processus à redémarrer montre que le protocole de sauvegarde de points de reprise non coordonnés force le redémarrage de tous les processus. Nous avons adapté le protocole afin de l'utiliser sur des groupes de processus pour de limiter le nombre de processus effectuant un retour arrière. L'analyse avec l'outil décrit dans le Chapitre 4 montre qu'il est possible de créer des groupes de processus tels que le volume des messages à enregistrer est d'environ 5% tout en ayant environ la moitié des

processus qui reviennent en arrière. Il est donc possible d'éviter un redémarrage global contrairement à un protocole de sauvegarde de points de reprise coordonnés.

Dans le chapitre suivant, nous décrivons un autre protocole de tolérance aux fautes, fondé sur les groupes de processus, qui minimise les effets d'une défaillance à un seul groupe.

Chapitre 6

HydEE : Un protocole de recouvrement arrière hiérarchique

Sommaire

6.1 Introduction **83**
6.2 Description **84**
6.2.1 Principes du protocole 84
6.2.2 Défaillance et redémarrage 84
6.2.3 Suppression des données obsolètes 92
6.2.4 Processus de redémarrage distribué 92
6.2.5 Preuve 92
6.3 Évaluation **95**
6.3.1 Description du prototype 96
6.3.2 Configuration 96
6.3.3 Performances en exécution sans faute 97
6.3.4 Évaluation des performances en redémarrage 99
6.4 Conclusion **100**

6.1 Introduction

Dans ce chapitre, nous proposons un nouveau protocole de recouvrement arrière hiérarchique pour les applications à émissions déterministes utilisant des groupes de processus appelé *HydEE*.

HydEE applique un protocole de sauvegarde de points de reprise coordonnés au sein des groupes et un protocole à enregistrement de messages pour les communications inter-groupe. Ce protocole permet d'éviter le retour arrière de l'ensemble des processus de l'application après une défaillance tout en limitant le nombre de messages à enregistrer. Ainsi, contrairement au protocole proposé dans le Chapitre 5, après la défaillance d'un processus, seuls les processus du même groupe que le processus fautif ont besoin d'effectuer un retour arrière car : 1) les messages nécessaires au redémarrage du groupe

ont été enregistrés ; 2) le déterminisme des émissions assure que l'état des autres groupes restera cohérent avec le groupe redémarrant.

Ce chapitre est organisé comme suit. Dans le paragraphe 6.2, nous présentons les principes et décrivons le pseudo-code de notre protocole. Nous prouvons dans ce même paragraphe que le protocole garantit que l'exécution est correcte après une défaillance. Nous présentons nos résultats expérimentaux dans le paragraphe 6.3.

6.2 Description

Dans ce paragraphe, nous commençons par décrire HydEE en fonctionnement normal dont l'algorithme est présenté sur la figure 6.3. Puis, nous décrivons la gestion des défaillances et notamment des causalités lors du redémarrage. L'algorithme des processus de l'application après une défaillance est présenté sur les figures 6.4 et 6.5. La figure 6.6 décrit l'algorithme du processus de redémarrage. Notons que le modèle utilisé ici est le même que celui décrit dans le paragraphe 5.2 du Chapitre 5.

6.2.1 Principes du protocole

Dans ce protocole, le système des processus est divisé en p groupes logiques. Un protocole de sauvegarde de points de reprise coordonnés au sein des groupes et un protocole à enregistrement de messages fondé sur l'émetteur entre les groupes (lignes 8-9 de la figure 6.3) sont utilisés. Chaque fois qu'un processus envoie un message à un processus appartenant à un autre groupe, il enregistre le contenu du message dans sa mémoire. Il sauvegarde également (dans sa mémoire) la date d'émission du message afin d'identifier les messages enregistrés à renvoyer en cas de défaillance. Du côté du récepteur, le processus sauvegarde, dans une table appelé RPP (décrite dans le paragraphe 5.3.4 du Chapitre 5), la date d'émission des messages pour chaque canal (ligne 14 de la figure 6.3). Cette information est utilisée afin de calculer la liste des messages orphelins pendant la phase de redémarrage. La date d'un processus est incrémentée à chaque émission ou réception d'un message (lignes 21-7 de la Figure 6.3).

Au sein d'un groupe, n'importe quel protocole de sauvegarde de points de reprise coordonnés peut être utilisé (bloquant ou non bloquant). En effet, étant donné que les messages inter-groupe sont enregistrés, il n'est pas nécessaire de les inclure dans le calcul de l'état global car ils seront rejoués. En d'autres termes, il n'est pas nécessaire d'empêcher la réception d'un message inter-groupe ni de les inclure dans le calcul de l'état global durant la phase de sauvegarde des points de reprise car ces messages sont enregistrés au niveau de l'émetteur. Au moment de la sauvegarde du point de reprise sur support stable, le processus y inclut sa phase et sa date actuelles, les messages enregistrés (avec leur date) et la table RPP (lignes 23-26 de la Figure 6.3).

6.2.2 Défaillance et redémarrage

Même si le déterminisme des émissions garantit que l'ordre de réception des messages n'a pas d'effets sur l'exécution, les messages sont partiellement ordonnés par leur ordre

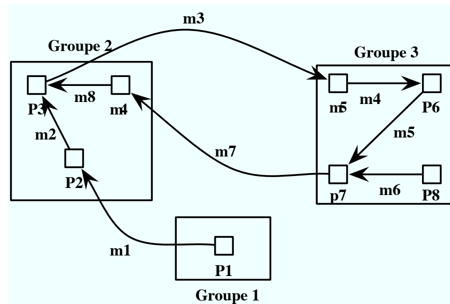


FIGURE 6.1 – Groupes de processus

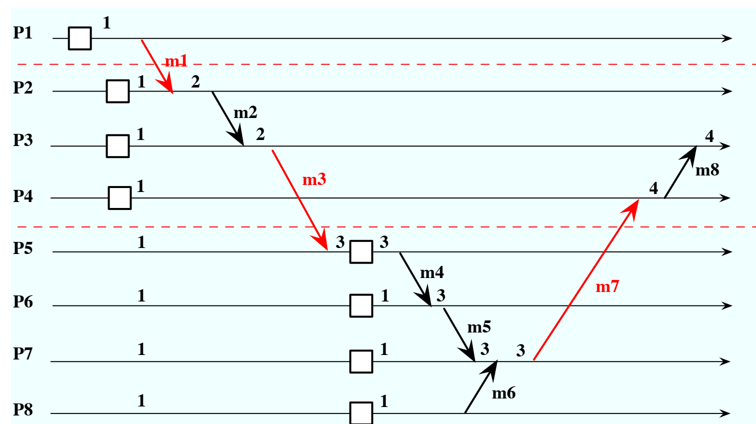


FIGURE 6.2 – Représentation temporelle

causal. En d'autres termes, si l'émission d'un message m' dépend de la réception d'un message m, HydEE doit assurer que durant le redémarrage, m' n'est pas renvoyé avant que m ne soit reçu. Afin d'illustrer le problème, la figure 6.1 présente une exécution avec 8 processus divisés en 3 groupes. La figure 6.2 est la représentation temporelle de la figure 6.1. Sur ces figures, les indices des messages représentent leur ordre causal. Dans une exécution sans faute, les messages m_3 et m_7 ne peuvent être émis avant la réception de m_1 et m_3 respectivement. Après la défaillance d'un processus du $Groupe_2$, tous les processus de ce groupe effectuent un retour arrière vers leur dernier point de reprise. m_3 devient donc un message orphelin. Grâce au déterminisme des émissions, les processus du $Groupe_3$ ne font pas de retour arrière pour recevoir le message m_3. Le message enregistré m_7 pourrait donc être renvoyé avant m_1 et m_8 pourrait être reçu avant m_2. Cependant, étant donné que m_8 dépend de m_2, l'exécution ne serait plus cohérente.

La différence entre les deux exécutions réside dans l'existence des messages orphelins : les processus du $Groupe_3$ n'ont pas besoin de recevoir m_3 pour envoyer leurs messages enregistrés. Ainsi, quand un processus reçoit un message provenant d'un processus d'un autre groupe, ce message peut devenir un message orphelin. Tous les processus dans un groupe doivent savoir que les messages qu'ils renvoient durant le redémarrage, qui

Variables Locales:
1: P_i, $Date_i$, $Phase_i$ /* L'identifiant, la date et la phase du processus i. $Phase_i$ est initialisée à 1 */
2: $Groupe_i$ /* L'identifiant du groupe auquel le processus i appartient */
3: $Enregistres_i \leftarrow \emptyset$ /* Ensemble des messages enregistrés par P_i */
4: $RPP_i \leftarrow [\perp, ..., \perp]$ /* $RPP_i[j]$ contient l'information de communication entre le processus P_i et le processus P_j appartenant à un autre groupe.$RPP_i[j]$.Maxdate est la date d'émission du dernier message reçu de P_j. $RPP_i[j][date]$ est la phase du message émis par P_j dont la date est $date$ */
5: $RM_i \leftarrow [faux, ..., faux]$ /* Tableau utilisé pour supprimé les données obsolètes. $RM_i[j] = vrai$ si le processus P_i doit envoyer un acquittement au processus P_j et $faux$ sinon */

6: **À l'envoi du message** msg **au processus** P_j
7: $Date_i \leftarrow Date_i + 1$
8: **si** $groupe_i \neq groupe_j$ **alors**
9: $Enregistres_i \leftarrow Enregistres_i \cup (P_j, Date_i, Phase_i, msg)$
10: Envoyer (msg, $Date_i$, $Phase_i$, $Groupe_i$) au processus P_j

11: **À la réception de (**msg**,** $Date_{emission}$**,** $Phase_{emission}$**,** $Groupe_{emission}$**) de** P_j
12: **si** $Groupe_{emission} \neq Groupe_i$ **alors**
13: $Phase_i \leftarrow Max(Phase_i, Phase_{emission} + 1)$
14: $RPP_i[j].Maxdate \leftarrow Date_{emission}$
15: $RPP_i[j][date_{emission}].phase \leftarrow Phase_{emission}$
16: **si** $RM_i[j] = vrai$ **alors**
17: Envoyer (***Ack***, $Date_{emission}$) à P_j
18: $RM_i[j] \leftarrow faux$
19: **sinon**
20: $Phase_i \leftarrow Max(Phase_i, Phase_{emission})$
21: $Date_i \leftarrow Date_i + 1$
22: Délivrer msg à l'application

23: **Au point de reprise du** $Groupe_i$
24: Coordonner les processus du $Groupe_i$
25: Sauvegarder ($ImagePs_i$, RPP_i, $Enregistres_i$, $Phase_i$, $Date_i$) sur support stable
26: $RM_i \leftarrow [vrai, ..., vrai]$

27: **À la réception de (*ACK*,** $MaxDate$**) du processus** P_j
28: **pour tout** $date \in Enregistres_i[P_j]$ **telle que** $date < MaxDate$ **faire**
29: Supprimer (P_j, $date$, $Phase$, $Groupe$, msg) de $Enregistres_j$

FIGURE 6.3 – Algorithme du protocole pour les processus de l'application en exécution sans faute

dépendent de messages orphelins, peuvent mener à une exécution non cohérente.

Une première idée pour gérer l'ordre causal entre les messages est d'enregistrer les déterminants des messages inter-groupe en mémoire. Ainsi, lors du rejeu, un message enregistré sera toujours reçu au moment adéquat. Cependant, étant donné que le protocole utilisé au sein des groupes est un protocole coordonné, après une défaillance et le retour arrière des processus, toutes les données sont perdues. Ainsi, il n'est plus possible d'ordonner les messages rejoués. Sur la figure 6.2, le déterminant du message m_1 est $(p_1, 1, p_2, 1)$ et le déterminant du message m_7 est $(p_7, 3, p_4, 1)$. Ces deux messages peuvent être reçus tout de suite après le redémarrage car ce sont les premiers messages reçus par p_2 et p_4, permettant ainsi l'envoi de m_2 et de m_8 alors que m_8 dépend de m_2. Ainsi, sauvegarder seulement les déterminants des messages enregistrés n'est pas une solution suffisante. L'autre solution serait d'enregistrer les déterminants de tous les messages (intra et inter-groupe), mais comme expliqué quand le Chapitre 2, la gestion et la sauvegarde des déterminants est coûteuse.

Afin d'assurer l'ordre causal lors du redémarrage, la méthode utilisé dans le Chapitre 5 est adaptée : utiliser des numéros de phases pour indiquer qu'un message dépend d'un potentiel orphelin.

Un message m' dépendant d'un message m qui vient d'un autre groupe devrait avoir un numéro de phase supérieur à celui de m. Durant le redémarrage, m' ne sera pas renvoyé tant que tous les messages orphelins à une phase inférieure à la sienne ne sont pas rejoués.

Les phases sont mises en œuvre de cette manière : chaque processus se voit attribuer un numéro de phase qui est attaché à chaque message qu'il envoie. La phase du processus est mise à jour chaque fois qu'il reçoit un message, comme suit :

1. si le message vient du même groupe et qu'il a une phase supérieure, le processus prend la phase du message (ligne 20 de la figure 6.3).
2. si la phase vient d'un autre groupe, il prend le maximum entre la phase du message incrémentée de 1 et sa propre phase (ligne 13 de la figure 6.3).

Les phases des messages reçus sont sauvegardées dans la table *RPP* afin d'identifier les phases des messages orphelins après une défaillance (lignes 14-15 de la figure 6.3). Afin d'illustrer comment les phases sont utilisées, prenons l'exemple de la Figure 6.2. Les phases de tous les processus sont initialisées à 1. Lorsque le processus p_2 reçoit le message m_1 (dont le numéro de phase est 1), sa phase devient 2 et le processus p_3 positionne sa phase à 2 à la réception du message m_2. Quand le processus p_5 reçoit le message m_3, sa phase devient 3 et les processus p_6 et p_7 mettent à jour leur phase à 3 après la réception des messages m_4 et m_5. La phase du message m_7 devient donc 3. Après la défaillance des processus du $Groupe_2$, le message enregistré m_1 est le seul qui peut être envoyé étant donné qu'il est le seul message dont la phase est inférieure à celle du message orphelin m_3 (2). Une fois que le message m_3 est rejoué, le message m_7 peut être renvoyé. L'ordre causal est ainsi garanti.

Afin d'assurer que les messages sont rejoués selon l'ordre causal après une défaillance, un processus appelé *processus de redémarrage* est utilisé. Il assure qu'un message intergroupe ne peut être réémis tant qu'il y a des messages orphelins à une phase inférieure

à la sienne. Contrairement au protocole décrit dans le Chapitre 5, la règle de rejeu est la même quelle que soit la nature du message.

Variables Locales:
1: P_i, $Date_i$, $Phase_i$, $Groupe_i$
2: $Enregistres_i \leftarrow \emptyset$
3: $OrphelinDate \leftarrow [\perp, ..., \perp]$ /* $OrphelinDate[j]$ est la date du dernier orphelon reçu par le processus P_j */

4: **À la défaillance du processus** P_i
5: Récupérer la dernière ($ImagePs_i$, RPP_i, $Enregistres_i$, $Phase_i$, $Date_i$) du support stable
6: **pour tout** $P \in Groupe_i$ **faire**
7: Redémarrer de $ImagePs$
8: Envoyer(***Redemarraee***, $Date_i$, $Groupe_i$) à tous les processus $P_k \notin Groupe_i$
9: Envoyer(***Ma_Phase***, $Phase_i$) au processus de redémarrage
10: **Attendre jusqu'à** réception de (***DerniereDate***, *date*) de tous les $P_k \notin Groupe_i$ et de (***NotifierEnvoiMsg***, $phase_i$) du processus de redémarrage

11: **À la réception de (*DernierеDate*, *date*) du processus** P_j
12: $OrphelinDate[j] \leftarrow DernierеDate$

13: **À l'envoi du message** msg **au processus** P_j
14: $Date_i \leftarrow Date_i + 1$
15: **si** $Groupe_i \neq Groupe_j$ **alors**
16: **si** $Date_i \leq OrphelinDate_i[P_j]$ **alors**
17: Envoyer (***OrphelinNotification***, $Phase_i$) au processus de redémarrage
18: **sinon**
19: Envoyer (msg, $Date_i$, $Phase_i$) au processus P_j
20: $Enregistres_i \leftarrow Enregistres_i \cup$ (P_j, $Date_i$, $Phase_i$, msg)
21: **sinon**
22: Envoyer (msg, $Date_i$, $Phase_i$) au processus P_j
23: **si** $\forall P_k \notin Groupe_i$, $Date_i > OrphelinDate_i[P_k]$ **alors**
24: Revenir à la focntion de l'Algorithme

25: **À la réception de (**msg, $Date_{send}$, $Phase_{send}$**) du processus** P_j
26: Utiliser la fonction de l'Algorithm 1

27: **À la sauvegarde du point de reprise dans le** $Groupe_i$
28: Utiliser la fonction de l'Algorithm 1

FIGURE 6.4 – Algorithme du protocole pour les processus de l'application en redémarrage

Lorsqu'une défaillance se produit, tous les processus appartenant au même groupe que le processus fautif font un retour arrière vers leur dernier point de reprise. Les processus d'un groupe ayant fait un retour arrière informent les processus des autres groupes en envoyant un message contenant la date sauvegardée dans le point de reprise (ligne 8 de la figure 6.4). Ils envoient aussi la phase à partir de laquelle ils redémarrent (celle contenue dans le point de reprise) au processus de redémarrage (ligne 17 de la figure 6.4). Cette dernière donnée est utilisée dans le cas où les processus de deux groupes différents subissent une défaillance simultanée. Dans ce cas, les messages inter-groupe qui ne sont pas des orphelins doivent également obéir aux règles des phases pour garantir l'ordre causal. Sur la figure 6.2, si les processus du $Groupe_2$ et $Groupe_3$ subissent une défaillance, le renvoi du message m_7 n'est plus géré par le processus de redémarrage

```
Variables Locales:
 1: P_i, Date_i, Phase_i, Groupe_i
 2: DateRedem ← [⊥, ..., ⊥]   /* DateRedem[j] est la date à partir de laquelle le processus P_j redémarre */
 3: OrphPhases_i ← ∅          /* Phases auxquelles le processus P_i a reçu un message orphelin */
 4: EnrRenvoyes_i ← ∅         /* Ensemble des messages enregistrés à renvoyer */
 5: enregistres_phase ← ∅     /* Ensemble des phases des messages enregistrés */

 6: À la défaillance d'un processus P_j ∉ Groupe_i
 7:    Attendre jusqu'à réception de (Redémarrage, Date_redem) de tous les processus P_k ∈ Groupe_j
 8:    pour tout P_k ∈ Groupe_j faire
 9:       Envoyer(DernièreDate, RPP_i[j].MaxDate) à P_k
10:       pour tout (P_k, Date, Phase, msg) ∈ Enregistres_i tel que Date > Date_redem faire
11:          Ajouter (P_k, Date, Phase, msg) à EnrRenvoyes_i
12:          enregistres_phase_i ← enregistres_phase_i ∪ phase
13:       pour tout date ∈ RPP_i[k] telle que date > Date_redem faire
14:          Ajouter RPP_i[k][date].phase to OrphPhases_i
15:    Envoyer(MsgEnregistres, enregistres_phase_i) au processus de redémarrage
16:    Envoyer(Orphan, OrphPhases_i) au processus de redémarrage
17:    Envoyer(Ma_Phase, Phase_i) au processus de redémarrage
18:    Attendre jusqu'à réception de (NotifierEnvoiMsg, Phase) du processus de redémarrage
19:    Utiliser les fonctions de l'Algorithme 1

20: À la réception de (NotifierEnvoiEnr, Phase_notif) du processus de redémarrage
21:    pour tout (P, Date, Phase, msg) ∈ EnrRenvoyes_i tel que Phase ≤ Phase_notif faire
22:       Envoyer (msg, Date, Phase, Cluster_i) à P
```

FIGURE 6.5 – Algorithme du protocole pour les processus de l'application non redémarrés

étant donné que ce message n'existe plus. Il est donc possible que les messages m_4, m_5, m_6 et m_7 soient régénérés avant le rejeu du message orphelin m_3. Pour les mêmes raisons que celles décrites précédemment, il est donc possible de confondre les message m_2 et m_8. Pour résoudre ce problème, un processus en redémarrage n'envoie un message que lorsqu'il n'y a plus d'orphelins à une phase inférieure à la sienne (ligne 10 de la figure 6.4). Sur la figure 6.2, étant donné que la phase du processus p_5 sauvegardée dans son dernier point de reprise est 3, il ne pourra envoyer le message m_4 que lorsque le message m_3 (dont la phase est 2) est reçu. Les messages qui sont causalement dépendants de m_4 ne peuvent donc pas être générés et émis avant sa réception. Notons que le processus p_8 peut envoyer le message m_6 car sa phase est égale à 1 au redémarrage. Cela ne pose pas de problème étant donné que m_6 ne dépend pas de m_3 et que l'ordre de réception de m_5 et m_6 n'aura pas d'impact sur l'émission de m_7.

Lorsqu'un processus p reçoit la notification de retour arrière d'un processus q, il calcule la liste des orphelins qu'il va recevoir de q en utilisant la table RPP : chaque message dans RPP qui a une date supérieure à la date contenue dans la notification est un message orphelin (lignes 13-14 de la figure 6.5). Il calcule également la liste des messages enregistrés qu'il doit renvoyer sur le canal (lignes 10-11 de la figure 6.5). Enfin, il envoie à q un message contenant la date du dernier message qu'il a reçu de lui (ligne 9 de la figure 6.5). Cette information est utilisée par q pour savoir si un message qu'il envoie pendant son redémarrage est un message orphelin ou pas (lignes 16-17 de la figure 6.4).

Étant donné qu'un processus ne peut envoyer de nouveaux messages tant qu'il y a des messages orphelins à une phase inférieure à la sienne (ligne 18 de la figure 6.5), il envoie sa phase courant au processus de redémarrage (ligne 17 de la figure 6.5). En effet, si l'on considère que dans la figure 6.2, le message m_7 n'est pas un message enregistré mais un message qui n'est pas encore envoyé, le processus p_7 ne peut l'envoyer tant que le message orphelin m_3 n'est pas reçu, sinon le même problème de causalité se poserait.

Une fois qu'un processus a reçu toutes les notifications de redémarrage des processus du groupe effectuant un retour arrière, il envoie les phases des messages enregistrés qu'il doit renvoyer (lignes 10-12, 15 de la figure 6.5) et le nombre de messages orphelins qu'il attend dans chaque phase (lignes 13-14, 16 de la figure 6.5) au processus de redémarrage. Le processus de redémarrage utilise la première information pour savoir quel processus notifier pour chaque phase, et la seconde information pour savoir combien d'orphelins il y a dans chaque phase.

Une fois que le processus de redémarrage a reçu tous les messages des processus de l'application, il commence la phase notification pour les phases pour lesquelles il n'y a pas d'orphelins à une phase inférieure (lignes 14-15 de la figure 6.6). Chaque fois qu'un processus ayant effectué un retour arrière doit envoyer un message orphelin, il envoie une notification au processus de redémarrage au lieu du message lui même (lignes 16-17 de la figure 6.4). Cette notification contient le numéro de phase du message. Lorsque toutes les notifications pour une phase ρ ont été reçues (lignes 14-15 de la figure 6.6), les notifications $NotiferEnvoiEnr$ (renvoi des messages enregistrés) et $NotifierEnvoiMsg$ (envoi des messages par les processus en redémarrage ou des nouveaux messages par les autres processus) pour la prochaine phase sans orphelins sont envoyées (lignes 17-20 et

Variables Locales:

1: $NbOrphPhase \leftarrow \emptyset$ /* $NbOrphPhase[phase]$ est le nombre de processus attendant un message orphelin à la phase $phase$ */
2: $Processus_phases \leftarrow \emptyset$ /* $Processus_phases[phase]$ est l'ensemble des processus dont la phase courante est $phase$ */
3: $MsgEPhase \leftarrow [\perp, ..., \perp]$ /* $MsgEPhase[phase]$ est l'ensemble des processus qui ont au moins un message enregistré dont la phase est $phase$ */

4: **À la réception de (*MsgEnregistres*,** $enregistres_phase$**) du processus** P_j
5: **pour tout** $phase \in enregistres_phase$ **faire**
6: $MsgEPhase[phase] \leftarrow MsgEPhase[phase] \cup P_j$

7: **À la réception (*Orphelin*,** $OrphPhases$**) du processus** P_j
8: **pour tout** $phase \in OrphPhases$ **faire**
9: $NbOrphelinPhase[phase] \leftarrow NbOrphelinPhase[phase] + 1$

10: **À la réception de (*Ma_Phase*,** $Phase_j$**) du processus** P_j
11: $Processus_phase[phase_j] \leftarrow Processus_phase[phase_j] \cup Pj$

12: **À la réception de (*OrphelinNotification*,** $OrphPhases_j$**) du processus** P_j
13: $NbOrphelinPhase[phase] \leftarrow NbOrphelinPhase[phase] - 1$
14: **si** $NbOrphelinPhase[phase] == 0$ **alors**
15: Démarrer **NotifierPhases**

16: **NotifierPhases**
17: **pour tout** $phase \in MsgEPhase$ **telle que** $\nexists phase' < phase \wedge NbOrphelinPhase[phase'] > 0$ **faire**
18: **pour tout** $p \in MsgEPhase[phase]$ **faire**
19: Envoyer (***NotifierEnvoiEnr***, $phase$) au processus p
20: Supprimer $phase$ de $MsgEPhase$
21: **pour tout** $phase \in Processus_phase$ **telle que** $\nexists phase' < phase \wedge NbOrphelinPhase[phase'] > 0$ **faire**
22: **pour tout** $p \in Processus_phase[phase]$ **faire**
23: Envoyer (***NotifierEnvoiMsg***, $phase$) au processus p
24: Supprimer $phase$ de $Processus_phase$

Figure 6.6 – Algorithme du protocole pour le processus de redémarrage

lignes 21-23 de la figure 6.6).

6.2.3 Suppression des données obsolètes

Dans un protocole de sauvegarde de points de reprise coordonnés, seul le dernier point de reprise sauvegardé sur support stable est nécessaire. Les points de reprise sont donc supprimés au fur et à mesure.

Lorsque les processus d'un groupe sauvegardent un nouveau point de reprise, les messages enregistrés par les processus des autres groupes, et reçus avant ce point de reprise, ne sont plus nécessaires. Pour pouvoir supprimer ces messages, après avoir pris un point de reprise, chaque processus répond par un acquittement au premier message de chaque processus des autres groupes (lignes 17-18 de la figure 6.3), permettant au processus recevant l'acquittement de supprimer les messages destinés à ce processus dont la date est antérieure à celle contenue dans l'acquittement (ligne 27-29 de la figure 6.3).

6.2.4 Processus de redémarrage distribué

Si le nombre de processus ou le nombre de notifications est élevé, un seul processus de redémarrage peut ne plus être suffisant étant donné qu'il doit recevoir les notifications de tous les messages orphelins et notifier tous les autres processus. Nous présentons ici l'algorithme d'un processus de redémarrage distribué. La figure 6.7 présente l'algorithme des processus de redémarrage. Nous ne présentons pas l'algorithme des processus de l'application car celui-ci reste inchangé.

L'idée est d'avoir plusieurs processus de redémarrage. Chacun s'occupe d'un sous-ensemble des processus de l'application. La différence par rapport à l'algorithme décrit dans la figure 6.6 réside dans la gestion des notifications des messages orphelins. En effet, étant donné que chaque processus de redémarrage s'occupe d'un sous-ensemble des processus de l'application, il se peut que des processus de redémarrage différents reçoivent des notifications de messages orphelins pour une même phase (étant donné que chacun gère un ensemble de processus). Il faut donc synchroniser les processus de redémarrage afin d'éviter de violer les causalités.

Lorsqu'un processus de redémarrage reçoit toutes les notifications d'orphelin des processus dont il s'occupe pour une phase, il en informe les autres processus de redémarrage (lignes 29-30 de la figure 6.7). Puis, lorsqu'un processus de redémarrage reçoit ce message de tous les autres processus de redémarrage pour une phase, il commence la phase de notification (ligne 33-34 de la figure 6.7).

6.2.5 Preuve

Sous l'hypothèse du déterminisme des émissions, une exécution correcte est une exécution telle que :

- Chaque processus envoie une séquence valide de messages.
- L'ordre causal entre les réceptions des messages est respecté.

Nous prouvons que HydEE garantit qu'une exécution reste correcte malgré plusieurs défaillances. Pour ce faire, nous considérons l'état à partir duquel les processus

Variables Locales:
1: $NbOrphPhase \leftarrow \emptyset$ /* $NbOrphPhase[phase]$ est le nombre de processus attendant un message orphelin à la phase *phase* */
2: $Processus_phases \leftarrow \emptyset$ /* $Processus_phases[phase]$ est l'ensemble des processus dont la phase courante est *phase* */
3: $MsgEPhase \leftarrow [\perp, ..., \perp]$ /* $MsgEPhase[phase]$ est l'ensemble des processus qui ont au moins un message enregistré dont la phase est *phase* */
4: $NbpsDeRedem$ /* Nombre de processus de redémarrage */
5: $MesOrphelins \leftarrow \emptyset$ /* $MesOrhelins$ est l'ensemble des phases pour lesquelles le proessus de redémarrage attend un orphelin */
6: $NbOrphelinRedem \leftarrow [\perp, ..., \perp]$ /* $NbOrphelinRedem$ est initialisée lorsque le processus de redémarrage reçoit les listes des orphelins de ses processus. A ce moment, $NbOrphelinRedem[phase] \leftarrow NbpsDeRedem$ */

7: **À la réception de (*MsgEnregistres*,** *enregistres_phase*) **du processus** P_j
8: **pour tout** $phase \in Enregistres_phase$ **faire**
9: $MsgEPhase[phase] \leftarrow MsgEPhase[phase] \cup P_j$

10: **À la réception (*Orphelin*,** *OrphPhases*) **du processus** P_j
11: **pour tout** $phase \in OrphPhases$ **faire**
12: $NbOrphelinPhase[phase] \leftarrow NbOrphelinPhase[phase] + 1$
13: $MesOrphelins \leftarrow MesOrphelins \cup phase$
14: $NbOrphelinRedem \leftarrow NbOrphelinRedem \cup phase$
15: $NbOrphelinRedem[phase] \leftarrow NbpsDeRedem$
16: **Attendre jusqu'à** réception de (***Orphelin***, *OrphPhases*) de tous les processus
17: Envoyer (***ListePhasesOrphelins***, *MesOrphelins*) à tous les processus de redémarrage

18: **À la réception de (textitListePhasesOrphelins,** *Liste*) **du processus de redémarrage** P_j
19: **pour tout** $phase \in NbOrphelinRedem$ **et** $phase \notin Liste$ **faire**
20: $NbOrphelinRedem \leftarrow NbOrphelinRedem - 1$
21: **pour tout** $phase \in Liste$ **et** $phase \notin NbOrphelinRedem$ **faire**
22: $NbOrphelinRedem \leftarrow NbOrphelinRedem \cup phase$
23: $NbOrphelinRedem[phase] \leftarrow NbpsDeRedem$
24: $NbOrphelinRedem[phase] \leftarrow NbOrphelinRedem[phase] - 1$

25: **À la réception de (textitMa_Phase,** $Phase_j$) **du processus** P_j
26: $Processus_phase[phase_j] \leftarrow Processus_phase[phase_j] \cup Pj$

27: **À la réception de (textitOrphelinNotification,** $OrphPhases_j$) **du processus** P_j
28: $NbOrphelinPhase[phase] \leftarrow NbOrphelinPhase[phase] - 1$
29: **si** $NbOrphelinPhase[phase] == 0$ **alors**
30: Envoyer (textit**OrphelinsRecus**, *phase*) à tous les processus de redémarrage

31: **À la réception de (textitOrphelinsRecus,** *Phase*) **du processus de redémarrage** P_j
32: $NbOrphelinRedem[phase] \leftarrow NbOrphelinRedem[phase] - 1$
33: **si** $NbOrphelinRedem[phase] == 0$ **alors**
34: Démarrer **Phase de notification**

35: **NotifierPhases**
36: **pour tout** $phase \in MsgEPhase$ **telle que** $\nexists phase' < phase \wedge NbOrphelinRedem[phase'] > 0$ **faire**
37: **pour tout** $p \in MsgEPhase[phase]$ **faire**
38: Envoyer (textit**NotifierEnvoiEnr**, *phase*) au processus p
39: Supprimer *phase* de $MsgEPhase$
40: **pour tout** $phase \in Processus_phase$ **telle que** $\nexists phase' < phase \wedge NbOrphelinRedem[phase'] > 0$ **faire**
41: **pour tout** $p \in Processus_phase[phase]$ **faire**
42: Envoyer (textit**NotifierEnvoiMsg**, *phase*) au processus p
43: Supprimer *phase* de $Processus_phase$

FIGURE 6.7 – Algorithme du Processus de Redémarrage distribué

redémarrent après une défaillance. Nous nous fondons sur les définitions et les lemmes présentés dans le Chapitre 5. Nous supposons l'existence d'un seul processus de redémarrage.

La preuve du premier point est la même que dans le Chapitre 5.

Prouver que chaque processus envoie une séquence valide de messages revient à prouver que tous les messages annulés sont rejoués. Afin d'assurer cela, le protocole doit garantir que ces messages ne sont pas perdus.

Les messages annulés sont soit des messages inter-groupe soit des messages intra-groupe. Les messages intra-groupe ne peuvent pas être perdus grâce aux points de reprise coordonnés. Après le redémarrage d'un état global cohérent, chaque processus va générer la même séquence de messages grâce au déterminisme des émissions. Étant donné que tous les messages inter-groupe sont enregistrés, il n'y a pas de messages perdus.

Les messages intra-groupe sont rejoués comme en exécution sans fautes. Les messages enregistrés sont rejoués lorsque tous les messages orphelins dans une phase inférieure à la leur ont été reçus (lignes 17-20 de la figure 6.6 et lignes 20-22 de la figure 6.5). Ainsi, tous les messages sont rejoués si tous les messages orphelins sont rejoués.

Lemme 6.1 *Tous les messages annulés sont renvoyés*

Preuve Soit $\mathscr{O}_{min_phase}$ l'ensemble des messages orphelins de plus petite phase.

Selon le Lemme 5.2, $\forall m_i \in \mathscr{R}_{<min_phase}$, m_i est envoyé.

Et $\forall m_j \in \mathscr{O}_{=min_phase}$, m_j est renvoyé selon la proposition 2.

Donc $\mathscr{O}_{min_phase} = \emptyset$ et $\forall m_i \in \mathscr{R}_{<min_phase+1}$, m_i est envoyé.

Et ainsi de suite jusqu'à ce que $\mathscr{O} = \emptyset$.

Nous prouvons maintenant que l'ordre causal est respecté. Afin de prouver cela, nous supposons deux messages annulés causalement dépendants et nous prouvons que le second ne peut pas être envoyé avant que le premier ne soit reçu. Ceci implique qu'un processus ne peut pas délivrer deux messages causalement dépendants dans un ordre incorrect. Nous démontrons d'abord quelques lemmes utiles pour la preuve.

On dénote par $Ph(m)$ la phase d'un message m.

Lemme 6.2 *Si $m \in \mathscr{O}$ est un message reçu par un processus P alors $Ph(m) < Ph(P)$.*

Preuve Étant donné que le protocole de sauvegarde de points de reprise ne crée pas de messages orphelins, m est un message inter-groupe. Donc $Ph(m) < Ph(P)$ selon la ligne 13 de la figure 6.3.

Lemme 6.3 *Soit $m \in \mathscr{O}$ un message orphelin et $m' \in \mathscr{R}$ un message tel que $m \to m'$. Si $\nexists m'' \in \mathscr{L}$ tel que $m \to m'' \to m'$, alors il y a un processus redémarrant de son dernier point de reprise sur la chaîne de causalités entre m et m'.*

Preuve Étant donné que $m \in \mathscr{O}$, le récepteur de m (dans le groupe A) n'a pas effectué de retour arrière avant la réception de m'. Étant donné que $m' \in \mathscr{R}$, son récepteur (dans le groupe B) a effectué un retour arrière avant sa réception.

- Si $A = B$, le lemme est trivialement démontré.
- Si $A \neq B$, il y a un message inter-groue m'' tel que $m \to m'' \to m'$. Étant donné que $m'' \notin \mathscr{L}$, l'émetteur de m'' a effectué un retour arrière avant son émission. Ainsi, la situation est la même que dans le premier cas.

Lemme 6.4 *Soit $m \in \mathscr{O}$ un message orphelin et m' un message tel que $m \to m'$. Si $m' \in \mathscr{R}$ alors m' ne peut pas être envoyé avant que m ne soit rejoué.*

Preuve Selon le Lemme 6.3, soit $\exists m'' \in \mathscr{L}$ tel que $m \to m'' \to m'$, soit il y a un processus P qui a redémarré d'un point de reprise sur le chaîne de causalités entre m'' et m.

- Si $\exists m'' \in \mathscr{L}$, selon le Lemme 6.2, $Ph(m) < Ph(m'')$. Selon les lignes 17-20 de la figure 6.6, m'' ne peut pas être envoyé avant que tous les messages orphelins dans une plus petite phase ne soient reçus.
- Si $\nexists m'' \in \mathscr{L}$, selon le Lemme 6.2, $Ph(m) < Ph(P)$. Selon les lignes 21-23 de la figure 6.6, P ne peut pas envoyer ses messages avant que tous les messages orphelins dans une phase plus petite que la sienne ne soient tous reçus.

Lemme 6.5 *Soient $m \in \mathscr{R}$ et $m' \in \mathscr{R}$ deux messages tels que $m \to m'$. Si $\forall m''$ tel que $m \to m'' \to m'$, $m'' \notin \mathscr{O}$ alors $m' \in \mathscr{S}$*

Preuve Nous prouvons cela par contradiction. Étant donné que $m' \in \mathscr{R}$, $m' \in \mathscr{L}$ ou $m' \in \mathscr{S}$. Supposons que $m' \in \mathscr{L}$. Étant donné que $m' \in \mathscr{L}$, alors l'émetteur de m' n'est pas revenu avant l'émission de m'. Cependant, étant donné que $m \in \mathscr{R}$ alors $\exists m''$ avec $m \to m'' \to m'$ tel que $m'' \in \mathscr{O}$ ce qui est une contradiction car nous avons supposé qu'il n'y a pas de messages orphelins entre m et m'.

Théorème 6.1 *Soient $m \in \mathscr{R}$ et $m' \in \mathscr{R}$ tels que $m \to m'$. m' n'est pas émis avant que m ne soit reçu.*

Nous prouvons cela selon l'existence d'un message orphelin entre m et m'

Preuve
- Si $\exists m'' \in \mathscr{O}$ tel que $m \to m'' \to m'$: Selon le Lemme 6.4, m' ne peut pas être émis tant que m'' n'est pas reçu. Étant donné que $m \in \mathscr{R}$, les messages sur la chaîne de causalité entre m et m'' sont rejoués de la même manière qu'une exécution sans faute : m'' ne peut pas être émis avant que m ne soit reçu. Ainsi, m' ne peut pas être envoyé avant que m ne soit reçu.
- Si $\nexists m'' \in \mathscr{O}$ tel que $m \to m'' \to m'$: Selon le Lemme 6.5, $m' \in \mathscr{S}$. La situation est équivalente à une exécution sans faute : m doit être reçu avant que m' ne puisse être envoyé.

6.3 Évaluation

HydEE a été mis en œuvre dans la bibliothèque MPICH2 (la même version que celle utilisée dans le Chapitre 4). Dans cette section, nous présentons les résultats expérimentaux obtenus. Nous commençons par décrire la configuration des expériences et notre

prototype. Puis nous présentons l'évaluation de HydEE en exécution sans faute et en redémarrage.

6.3.1 Description du prototype

Nous avons intégré HydEE dans MPICH2 de la même manière que le protocole décrit dans le Chapitre 5. La seule différence réside dans les données attachées aux messages. En effet, HydEE n'attache que le numéro de phase et la date sur chaque message de l'application. Nous n'avons pas intégré de protocoles de sauvegarde de points de reprise coordonnés pour les mêmes raisons que celles décrites dans le paragraphe 5.4.3 du Chapitre 5.

Le processus de redémarrage est un processus MPI supplémentaire qui est lancé par *Hydra* (service de gestion des processus dans MPICH2) lorsqu'une défaillance est détectée. Tous les processus de l'application sont connectés au processus de redémarrage en utilisant les fonctions $MPI_Comm_Connect()$ et $MPI_Comm_Accept()$. Ainsi, le processus de redémarrage peut communiquer avec les processus de l'application durant le redémarrage en utilisant les primitives de communication MPI. Cependant, les fonctions Connect/Accept ne fonctionnent que sur TCP dans la version actuelle de MPICH2. Nous ne présentons donc pas d'évaluation de HydEE en redémarrage sur Myrinet/MX. De plus, seul le processus de redémarrage centralisé a été mis en œuvre.

6.3.2 Configuration

Dans ce paragraphe, nous présentons la plate-forme utilisée pour réaliser les expériences. Nous exposons également les résultats de regroupement obtenus grâce à l'outil de partitionnement décrit dans le Chapitre 4. Le regroupement obtenu est celui utilisé pour l'évaluation des performances des applications.

6.3.2.1 Plate-forme

Les expériences ont été réalisées sur la plate-forme Grid'5000. Les évaluations sur Netpipe ont été réalisées sur 2 nœuds de la grappe de Lille équipés de 2 processeurs Intel Xeon E5440 QC (4 cœurs), 8 GO de mémoire et une interface Myri-10G 10G-PCIE-8A-C. Les évaluations des performances des applications en exécution sans fautes ont été réalisées sur 41 nœuds avec les mêmes caractéristiques et 25 nœuds équipés de 2 processeurs AMD Opteron 285 (2 cœurs), 4 GO de mémoire et une interface Myri-10G 10G-PCIE-8A-C.

L'évaluation du redémarrage a été réalisée sur le site de Lille en utilisant 33 nœuds équipés de 2 Intel Xeon E5440 QC (4 cœurs), 8 GO de mémoire et d'une carte réseau Myri-10G 10G-PCIE-8A-C. Un nœud a été spécialement dédié au processus de redémarrage.

Pour nos expériences, nous avons utilisé 6 applications de la suite des *NAS Parallel Benchmarks* [57] sur un problème de taille D sur 256 processus. Pour chaque résultat présenté, chaque application a été exécutée 5 fois et le meilleur temps d'exécution a été choisi.

6.3.2.2 Regroupement des processus dans les applications

La table 6.1 montre les résultats obtenus en exécutant l'algorithme décrit dans le Chapitre 4 avec Scotch avec la fonction de coût décrite dans le paragraphe 4.4.3.2 de ce même chapitre. Cette table présente, pour chaque application, le nombre de groupes, la taille du groupe auquel le processus 0 appartient, le ratio des processus qui redémarrent en cas de défaillance et la quantité de données enregistrée et totale échangée durant toute l'exécution. Nous avons présenté la taille du groupe auquel le processus 0 appartient car dans l'évaluation des performances en redémarrage, nous avons supposé que c'est le groupe auquel ce processus appartient qui subit une défaillance.

Les résultats montrent que pour toutes les applications sauf FT, l'outil parvient à trouver une configuration qui limite le nombre de processus à redémarrer après une défaillance à 30% en enregistrant moins de 20% des messages. FT ne présente pas de bons résultats car cette application utilise des primitives de communication *all-to-all.*

	Nb Groupes	Taille groupe Processus 0	Processus à Redémarrer	Qte Enr/Totale des données (en GO)
NPB BT	5	63	21.78%	143/791 (18.09%)
NPB CG	16	16	6.25%	440/2318 (18.98%)
NPB FT	2	129	50%	431/860 (50.19%)
NPB LU	8	32	12.5%	44/337 (13.26%)
NPB MG	4	64	25%	13/66 (19.63%)
NPB SP	6	32	18.56%	289/1446 (20.04%)

Table 6.1 – Groupes de processus dans les *NAS Parallel Benchmarks* sur 256 processus

6.3.3 Performances en exécution sans faute

Nous avons évalué les performances de HydEE en exécution normale sur Myrinet 10G. Nous commençons par présenter les performances des communications puis celles des applications.

6.3.3.1 Performances des communications

La figure 6.8 compare les performances des communications de la version native MPICH2 sur Myrinet 10G, aux performances de HydEE avec et sans enregistrement de messages. Les tests sont exécutés en utilisant l'application NetPipe [76]. Elle représente la latence et la réduction de bande passante en pourcentage de la version native de MPICH2. La figure 6.9 présente les performances de HydEE et de MPICH2.

Les résultats montrent que HydEE introduit un faible surcoût sur les performance des communications, et cela seulement sur les petits messages (en moyenne 9% sur la latence et 8% sur la bande passante sur les message d'une taille inférieure à $1Ko$). Afin d'expliquer la présence des deux pics, observons la figure 6.9. Les deux pics de la figure 6.8 correspondent aux deux plateaux sur la figure 6.9 (pour les messages de

taille égale à 32 octets pour MPICH2 et 19 octets pour HydEE). Sur cette figure, nous remarquons que HydEE atteint les plateaux avant MPICH2. Ceci est dû aux données supplémentaires que HydEE attache aux messages (phase et date d'émission).

Les performances avec et sans enregistrement de messages fondés sur l'émetteur sont équivalentes. Cela signifie que la technique d'enregistrement des messages utilisée n'a pas d'impact sur les performances et que le surcoût n'est dû qu'aux données supplémentaires attachées aux messages.

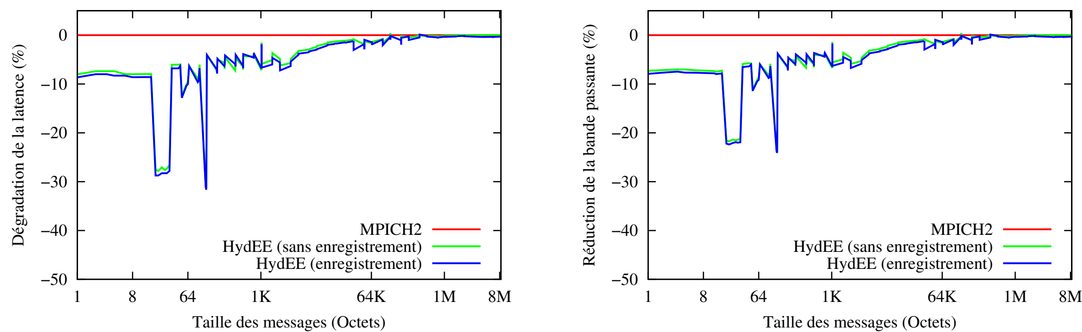


FIGURE 6.8 – Dégradation des performances de NetPipe sur Myrinet 10G

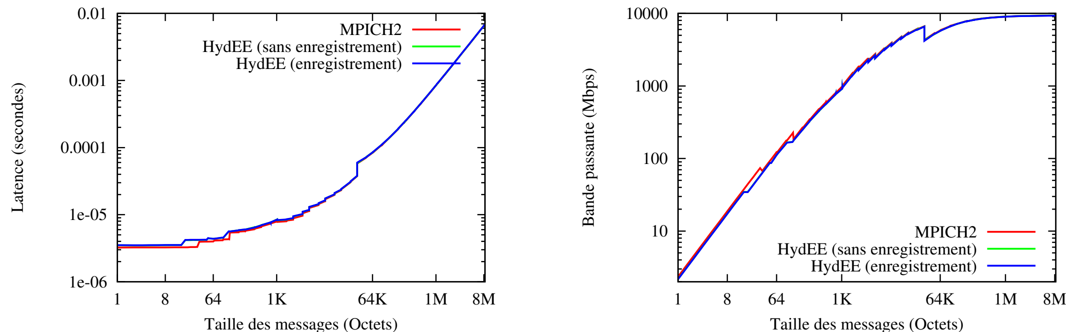


FIGURE 6.9 – Performances de Netpipe sur Myrinet 10G

Nous remarquons que les courbes d'évaluation des performances en communications décrites dans le Chapitre 5 sont similaires. Cependant HydEE offre de meilleurs résultats pour les petits messages car il attache moins de données aux messages.

6.3.3.2 Performances des applications

La figure 6.10 présente l'évaluation de HydEE en exécution sans faute en utilisant les *NAS Parallel Benchmarks*. Elle présente une comparaison entre les performances de HydEE (avec les groupes présentés dans le paragraphe 6.3.2.2), de MPICH2 et de l'enregistrement de tous les messages. Nous avons évalué les performances de l'enregistrement

de tous les messages (qui correspond à avoir un processus par groupe), pour montrer que le choix des groupes a un impact sur les performances. Les résultats sont normés par rapport au temps d'exécution. MPICH2 sans protocole de recouvrement arrière est choisi comme référence.

Les résultats montrent que les performances de HydEE sont presque équivalentes à celle de MPICH2. De plus, sur des applications où l'enregistrement de tous les messages introduit un surcoût (6% pour CG par exemple), les performances de HydEE sont quasiment équivalentes à celles de la version native de MPICH2. L'utilisation des groupes de processus permet donc d'améliorer les performances.

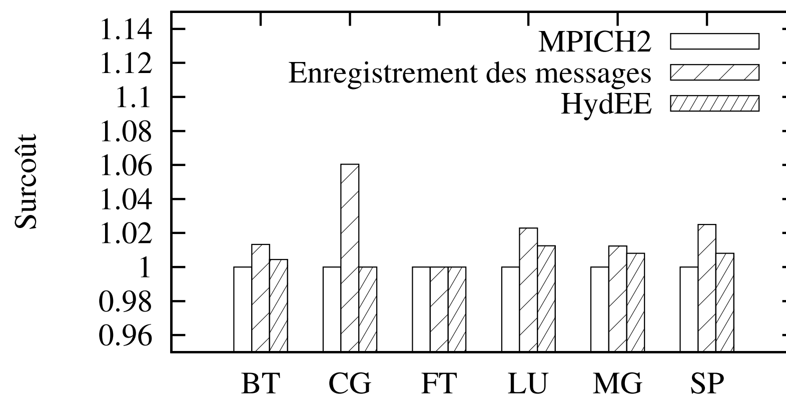


FIGURE 6.10 – Performances des *NAS Parallel Benchmarks* sur MX sur 256 processus

6.3.4 Évaluation des performances en redémarrage

Comme mentionné précédemment, l'évaluation des performances en redémarrage n'a été réalisée que sur TCP. De plus, le protocole coordonné au sein des groupes n'a pas encore été intégré au prototype. Cela ne pose pas de problème pour les raisons citées dans la paragraphe 5.4.3.

Afin d'évaluer les performances en redémarrage, nous avons procédé comme suit : nous avons exécuté l'application une première fois avec HydEE et les groupes obtenus dans le paragraphe 6.3.2.2 afin d'obtenir la liste des messages enregistrés et des messages orphelins. À la fin de cette exécution, les messages enregistrés sont copiés sur disque. Puis, nous exécutons l'application une seconde fois, en simulant le redémarrage des processus appartenant au même groupe que le processus dont l'identifiant est 0. Ceci signifie que seuls les processus de ce groupe s'exécutent réellement, les autres processus exécutent le protocole de redémarrage décrit sur la figure 6.5. Ils lisent les fichiers où les messages ont été enregistrés afin de récupérer la liste des messages à renvoyer et des messages orphelins, puis, ils renvoient les messages enregistrés selon les notifications reçues du processus de redémarrage.

La figure 6.11 présente l'évaluation de HydEE en recouvrement en utilisant les applications BT, CG, LU, MG et SP de la suite des *NAS Parallel Benchmarks*. Les résultats présentés sont normés par rapport au temps d'exécution. Le temps d'exécution en exécution sans fautes est pris comme référence.

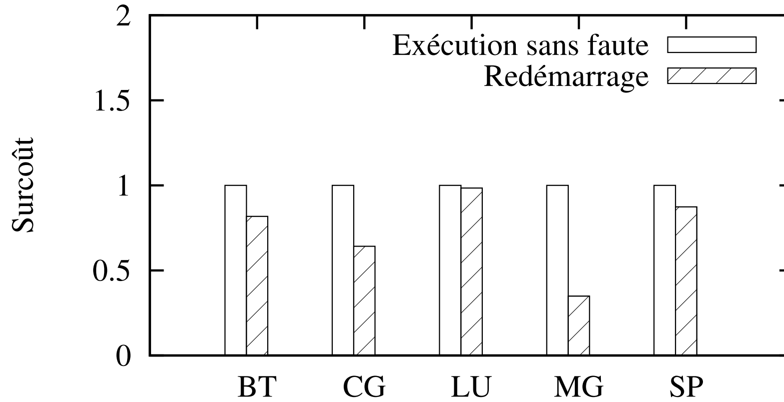


FIGURE 6.11 – Performances des *NAS Parallel Benchmarks* en Redémarrage sur 256 processus

Les résultats montrent que pour pour toutes les applications, HydEE améliore les performances en redémarrage par rapport à l'exécution sans fautes. Il y a deux raisons possibles à cette amélioration. Premièrement, les messages inter-groupe sont rejoués à partir des données enregistrées et ne sont pas régénérés par les processus. Ils peuvent donc être renvoyés dès que la notification correspondante est reçue du processus de redémarrage. Cela diminue la possibilité que les processus qui redémarrent aient à attendre ces messages. Deuxièmement, au lieu d'envoyer les messages orphelins aux processus de l'application, les processus en redémarrage envoient seulement une notification au processus de redémarrage. Ceci contribue également à la diminution du coût des communications.

6.4 Conclusion

Dans ce chapitre, nous avons présenté HydEE, un protocole hiérarchique pour les applications à émissions déterministes fondé sur les groupes de processus. En utilisant un protocole de sauvegarde de points de reprise coordonnés au sein des groupes et en enregistrant les messages inter-groupe sur la mémoire des émetteurs et grâce au déterminisme des émissions, HydEE permet de confiner les défaillances aux groupes où elles ont lieu sans nécessiter l'enregistrement de tous les déterminants contrairement aux protocoles de recouvrement arrière hiérarchiques existants. Seules les dates d'émission des messages enregistrés sont sauvegardées sur la mémoire de l'émetteur.

Nous avons présenté le protocole et ses différents algorithmes. Nous avons également

prouvé qu'il assure une exécution correcte en dépit des défaillances.

HydEE a été mis en œuvre dans la bibliothèque MPICH2 sur TCP, en mémoire partagée et sur réseau rapide Myrinet/MX. Les évaluations sur réseau haute performance montrent que le protocole a un faible impact sur la latence et la bande passante pour les messages de petite taille. Pour les grands messages, le protocole n'introduit aucun surcoût.

L'étude des performances en exécution normale des *NAS Parallel Benchmarks* montre que les performances de HydEE sont quasiment équivalentes à celles de la version native de MPICH2 et meilleures que celles de l'enregistrement de tous les messages.

L'évaluation du redémarrage montre que dans certains cas, l'exécution en redémarrage fournit de meilleures performances que l'exécution sans fautes.

Chapitre 7

Conclusion

Avec l'augmentation de la taille des machines parallèles, la tolérance aux fautes est devenue un enjeu important. En effet, sur les machines exascale et au delà, le temps moyen entre les défaillances ($MTBF$) devrait être de quelques heures seulement.

Parmi les techniques de tolérance aux fautes existantes, la tolérance aux fautes par recouvrement arrière semble rester la mieux adaptée. Il existe deux grandes familles de protocole de recouvrement arrière : les protocoles de sauvegarde de points de reprise et les protocoles à enregistrement de messages. Ces protocoles ne sont pas adaptés aux contraintes de telles architectures. En effet, ils forcent le redémarrage de tous les processus ou l'enregistrement de tous les messages.

Dans ce document, nous nous sommes intéressés aux applications MPI. Nous avons étudié leurs caractéristiques afin de proposer des protocoles de tolérance aux fautes qui prennent en considération ces caractéristiques.

Contributions

Dans ce document, nous avons étudié, dans une première partie, les caractéristiques des applications MPI du calcul haute performance. Puis, dans une seconde partie, nous avons proposé deux protocoles de tolérance aux fautes fondés sur ces caractéristiques.

Caractéristiques des applications MPI

Afin de mettre en œuvre de nouveaux protocoles de tolérance aux fautes, nous avons mis en avant deux caractéristiques des applications MPI : le déterminisme des émissions et l'existence de groupes de processus dans les schémas de communications de ces applications.

Déterminisme dans les applications MPI Les protocoles de tolérance aux fautes s'appuient sur des modèles d'exécution différents : les protocoles de sauvegarde de points de reprise supposent que les applications sont non déterministes alors que ceux à enregistrement de messages sont fondés sur le fait qu'elles sont déterministes par morceaux.

Afin de voir quel est le modèle le plus adapté aux applications de calcul haute performance, nous avons étudié le déterminisme dans les applications MPI. En analysant ces applications, nous avons mis en avant un nouveau type de déterminisme, que nous avons appelé "déterminisme des émissions". Dans une application à émissions déterministes, les messages sont toujours émis dans le même ordre pour le même ensemble de données en entrée. Du côté du récepteur, les messages peuvent être reçus dans n'importe quel ordre sans avoir d'effets sur les messages émis d'une exécution à une autre. Nous avons formalisé la définition du déterminisme des émissions et expliqué comment il peut être utilisé sur des applications MPI. Nous avons analysé 26 applications représentatives du calcul haute performance et avons constaté que la plupart d'entre elles sont à émissions déterministes.

Dans les protocoles de tolérance aux fautes, cette caractéristique a pour avantages :
- D'une part, de ne pas nécessiter la sauvegarde de déterminant car l'ordre de réception des messages n'a pas d'impact sur les messages émis.
- D'autre part, cette même raison permet d'éviter la création de message orphelin, permettant ainsi d'éviter de forcer le retour arrière de processus ayant reçu de tels messages.

Le déterminisme des émissions permet ainsi de concevoir une nouvelle famille de protocoles de tolérance aux fautes de façon à remédier aux problèmes des protocoles existants.

Groupes de processus dans les applications MPI La plupart des protocoles de tolérance aux fautes sont utilisés à plat : le même protocole est utilisé par tous les processus, sans prendre en considération les caractéristiques de la machine (processus s'exécutant sur différents cœurs d'un même nœuds) ou de l'application.

Nous avons étudié les schémas de communications dans les applications MPI afin d'observer comment les processus communiquent.

Dans un premier temps, nous avons étudié les schémas de communications des primitives de communications collectives, car celles-ci impliquent tous les processus. Les résultats ont montré que mis à part les schémas de communications engendrés par la primitive *MPI_ Alltoall*, les schémas créés par les algorithmes utilisés par les autres fonctions sont tels qu'il est possible de former des groupes de processus. Par la suite, nous avons analysé les schémas de communications d'applications couvrant 6 des 13 motifs de *Berkley*. L'analyse montre que pour certaines applications, les processus communiquent par groupe de façon quasi naturelle. Dans d'autres applications, l'existence des groupes n'est pas triviale.

En formant des groupes de processus, il est possible d'appliquer des protocoles différents au sein et entre les groupes. Ces groupes doivent cependant vérifier deux conditions :

1. Avoir une taille minimisée
2. Minimiser le volume des données échangées entre eux

Pour cela, nous avons développé un algorithme qui, à partir de la quantité de données échangées entre chaque paire de processus, crée des groupes de façon à minimiser ces

deux paramètres en utilisant une fonction de coût. Cette dernière dépend du protocole utilisé.

Les résultats de nos évaluations montrent que pour la plupart des applications, il est possible de former des groupes de processus en minimisant ces deux paramètres. Néanmoins, pour des applications telles que *FT* de la suite des *NAS Parallel Benchmarks*, les résultats obtenus montrent que l'utilisation de la primitive *MPI_ Alltoall* ne permet pas de nettement les minimiser. Cependant, l'utilisation de cette primitive devrait être minimisée sur des architectures exascale.

Nouveaux protocoles de tolérance aux fautes pour les applications à émissions déterministes

À partir de ces deux caractéristiques, nous avons proposé deux protocoles de tolérance aux fautes : un protocole de sauvegarde de points de reprise non coordonnés qui évite l'effet domino et un protocole hiérarchique fondés sur les groupes de processus qui limite la propagation d'une défaillance au groupe fautif.

Ces deux protocoles ont été mis en œuvre dans la bibliothèque MPICH2 sur TCP, en mémoire partagée et sur réseau rapide.

Un protocole de sauvegarde de points de reprise non coordonnés pour les applications à émissions déterministes Les protocoles de sauvegarde de points de reprise non coordonnés ont pour avantage de ne pas coordonner l'ensemble des processus pour la sauvegarde des points de reprise. Cependant, un de leur principal inconvénient est l'effet domino.

Nous avons proposé un protocole de sauvegarde de points de reprise non coordonnés pour les applications à émissions déterministes qui évite l'effet domino en n'enregistrant qu'un sous-ensemble des messages de l'application. Contrairement aux protocoles à enregistrement de messages, seules les dates d'émission des messages enregistrés sont sauvegardées sur la mémoire de l'émetteur. Lorsqu'une défaillance se produit, seuls les processus ayant envoyé un message à un processus fautif effectuent un retour arrière car le déterminisme des émissions garantit que les messages orphelins sont renvoyés.

Les évaluations que nous avons effectuées sur réseau haute performance montrent que le protocole n'introduit qu'un faible surcoût sur les performances des communications pour des messages de petite taille et aucun surcoût pour les messages de grande taille. De plus, l'évaluation des performances des applications montre que le protocole n'introduit qu'un faible surcoût comparé à la version native de MPICH2. L'évaluation du nombre de processus à redémarrer montre cependant que le protocole de sauvegarde de points de reprise non coordonnés force le retour arrière de tous les processus, car les processus communiquent tous les uns avec les autres, directement ou indirectement.

Nous avons adapté le protocole pour l'utiliser sur des groupes de processus afin de limiter le nombre de processus effectuant un retour arrière. L'évaluation obtenue en utilisant l'outil de partitionnement montre qu'il est possible de créer des groupes de processus de façon à enregistrer moins de 9% des messages tout en limitant le nombre de processus effectuant un retour arrière à environ 60%.

HydEE : Un protocole de recouvrement arrière hiérarchique Le protocole décrit précédemment limite le nombre de processus à redémarrer lorsqu'il est utilisé sur des groupes de processus, mais pas la propagation d'une défaillance aux processus des autres groupes. Afin de confiner les défaillances, nous avons, à partir des groupes de processus, proposé un protocole qui limite la propagation d'une défaillance aux processus appartenant au même groupe que le processus fautif.

Un protocole de sauvegarde de points de reprise coordonnés est utilisé au sein de groupes, et un protocole à enregistrement de messages fondé sur l'émetteur est utilisé pour les messages inter-groupe. Le déterminisme des émissions permet de n'enregistrer aucun déterminant contrairement aux protocoles de recouvrement arrière hiérarchiques existants. Seules les dates d'émission des messages enregistrés sont sauvegardées sur la mémoire des émetteurs. Lorsqu'une défaillance se produit dans un groupe, tous les processus (et seulement ces processus) de ce groupe effectuent un retour arrière vers leur dernier point de reprise. Le déterminisme des émissions permet de ne pas forcer le retour arrière d'autres processus car les messages orphelins sont renvoyés.

Les expériences effectuées montrent que ce protocole introduit un faible surcoût sur les performances des communications sur les petits messages et aucun surcoût sur les messages de grande taille. De plus, les performances des applications en exécution sans faute sont quasiment similaires aux performances de la version native de MPICH2. Les évaluations des performances en redémarrage montrent que celles-ci peuvent être meilleures que celles obtenues en exécution sans fautes.

La table 7.1 présente une comparaison quantitative entre le protocole décrit dans le Chapitre 5 adapté aux groupes de processus, HydEE, le protocole de sauvegarde de points de reprise coordonnés [9], le protocole à enregistrement de messages optimiste [35] et le protocole hiérarchique décrit dans [51].

	Coordination	Redémarrage global	Enregistrement des messages	Sauvegarde de tous les déterminants
Protocole Chapitre 5	Non	Non	Partiel	Non
HydEE	Partielle	Non	Partiel	Non
Protocole coordonné [9]	Totale	Oui	Non	Non
Protocole optimiste [35]	Non	Oui/Non	Total	Oui
Hiérarchique [51]	Partielle	Non	Partiel	Oui

Table 7.1 – Étude comparative des différents protocoles de tolérance aux fautes

Perspectives

Les travaux présentés dans ce documents ouvrent de nombreuses perspectives de recherche.

Dans un premier temps, nous travaillons sur la mise en œuvre du protocole de sauvegarde de points de reprise coordonnés. Celle-ci est ajoutée à *Hydra* (service de gestion des processus dans MPICH2). La sauvegarde des points de reprise est basée sur BLCR [77]. De plus, nous devons mettre en œuvre la copie des messages sur support stable afin d'évaluer complètement les deux protocoles décrits. Il faudra également mettre en œuvre le redémarrage partiel dans *MPICH2* qui n'existe pas encore. Cela permettra d'évaluer les performances réelles en redémarrage.

Évolution des schémas de communications dans le temps

Les schémas de communications utilisés pour former les groupes de processus ont été obtenus en additionnant la quantité des messages échangés entre les processus sur l'ensemble de l'exécution. Cependant, la façon avec laquelle les processus communiquent entre eux peut évoluer dans le temps. Ainsi, afin que les groupes créés reflètent réellement la manière dont les processus communiquent, il faudra détecter le moment où les processus changent leur façon de communiquer et adapter les groupes de processus à ce changement.

Simplification du redémarrage

Même si les deux protocoles proposés ici limitent le nombre de processus à redémarrer, le redémarrage peut être coûteux. En effet, les processus de redémarrage doivent rejouer les messages selon les numéros de phases afin de garantir que l'ordre causal des messages est respecté. Lorsque le nombre de processus augmente, l'utilisation d'un processus de redémarrage distribué devient nécessaire. Cependant, s'il y a beaucoup de phases et peu de messages par phase, ces processus devront gérer beaucoup de notifications.

L'unique possibilité de confondre deux messages est d'avoir un appel à une primitive de réception avec une source non nommée. Il faudra donc pouvoir identifier les messages pouvant être confondus. Une idée pour réaliser cela serait d'identifier les portions de code qui sont causalement dépendantes et d'attribuer à chaque portion un identifiant unique (un compteur par exemple) et d'ajouter cette identification dans les messages. Les messages dans une même partition (et donc non causalement dépendants) peuvent être reçus dans n'importe quel ordre grâce au déterminisme des émissions. Ces identifiants sont attribués par la bibliothèque MPI. Lorsqu'une défaillance se produit, il suffit simplement que le processus fautif respecte l'ordre de réception des messages selon l'identifiant des portions. De cette façon, l'utilisation des phases et donc des processus de redémarrage n'est plus nécessaire.

Intervalle de sauvegarde des points de reprise

Dans HydEE, un protocole de sauvegarde de points de reprise coordonnés est utilisé au sein des groupes. Cependant, il faudrait adapter la formule de calcul de l'intervalle de sauvegarde de points de reprise. En effet, les formules existantes supposent que les points de reprise sont sauvegardés sur tous les processus en même temps, alors que

dans HydEE, la coordination se fait seulement au sein des groupes et il n'y a pas de coordination entre eux.

La formule utilisée peut être la même que les formules existantes, mais appliquée à chaque groupe séparément. La formule à trouver servirait donc à savoir quel est le décalage optimal entre les sauvegardes de points de reprise au niveau des différents groupes afin de limiter les écritures simultanées des points de reprise entre les groupes.

Évaluation de la consommation d'énergie

Les protocoles proposés dans ce document ne forcent le redémarrage que d'un sous-ensemble des processus de l'application. Les autres processus peuvent se retrouver dans un état passif.

Si l'on considère qu'un processus qui s'exécute a une consommation d'énergie maximale, lorsqu'un sous-ensemble des processus redémarrent, ceux qui ne redémarrent pas vont avoir une consommation inférieure permettant ainsi une économie d'énergie. Ainsi, ces protocoles permettent de diminuer le coût en énergie après une défaillance par rapport à un protocole de sauvegarde de points de reprise coordonnés. Cependant, pour avoir une évaluation complète, il faudrait introduire également le passage des processus qui ne redémarrent pas d'un état en exécution à un autre état. Une idée serait de faire hiberner ces processus en attendant que ceux qui ont redémarré aient fini le recouvrement. Dans ce cas, il faudrait évaluer le coût nécessaire du passage d'un état en exécution vers un état en hibernation et inversement. Il est également possible de profiter des ressources libres afin d'exécuter une autre application et il faudrait donc évaluer l'effet sur la consommation d'énergie. De plus, les protocoles décrits dans ce document enregistrent certains messages, il faudrait donc inclure le coût de cette sauvegarde en évaluant combien d'énergie cela consomme.

Bibliographie

[1] Jack Dongarra, Pete Beckman, et al. The international exascale software roadmap. *International Journal of High Performance Computer Applications*, 25(1), 2011.

[2] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Redesigning the message logging model for high performance. *Concurr. Comput. : Pract. Exper.*, 22 :2196–2211, November 2010.

[3] Katie Antypas, John Shalf, and Harvey Wasserman. NERSC-6 Workload Analysis and Benchmark Selection Process. Technical Report LBNL-1014E, Lawrence Berkeley National Laboratory, Berkeley, 2008.

[4] Rolf Riesen. Communication Patterns. In *Workshop on Communication Architecture for Clusters CAC'06*, Rhodes Island, Greece, April 2006. IEEE.

[5] Graham E. Fagg and Jack Dongarra. Ft-mpi : Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, 2000. Springer-Verlag.

[6] The Sequoia Benchmarks. http ://asc.llnl.gov/sequoia/benchmarks/.

[7] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 38–, Washington, DC, USA, 2004. IEEE Computer Society.

[8] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34 :375–408, September 2002.

[9] K. Mani Chandy and Leslie Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1) :63–75, 1985.

[10] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[11] Yuval Tamir and Carlo H. Séquin. Error recovery in multicomputers using global checkpoints. In *In 1984 International Conference on Parallel Processing*, pages 32–41, 1984.

[12] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *In Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, 1992.

[13] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Inf. Process. Lett.*, 25 :153–158, May 1987.

[14] Richard Koo and Sam Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[15] Flaviu Cristian and Farnam Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 8th Symposium on Reliable Distributed Systems, SRDS '91*, pages 12–20, 1991.

[16] Parameswaran Ramanathan and Kang G. Shin. Use of common time base for checkpointing and rollback recovery in a distributed system. *IEEE Transactions on Software Engineering*, 19 :571–583, June 1993.

[17] Zhijun Tong, Richard Y. Kain, and W. T. Tsai. A low overhead checkpointing and rollback recovery scheme for distributed systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems, SRDS '89*, pages 12–20, 1989.

[18] Nuno Neves. Coordinated checkpointing without direct coordination. In *in Proceedings of the 3rd IEEE International Computer Performance and Dependability Symposium*, pages 23–31, 1998.

[19] B Bhargava and Lian Shu-Renn. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Seventh Symposium on Reliable Distributed Systems*, pages 3–12, Columbus, OH , USA, 1988.

[20] Yi-Min Wang. *Space reclamation for uncoordinated checkpointing in message-passing systems*. PhD thesis, Champaign, IL, USA, 1993. UMI Order No. GAX94-11816.

[21] Daniele Briatico, Augusto Ciuffoletti, and Luca Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 207–215, Silver Spring (Maryland), October 1984.

[22] Jean-Michel Helary, Achour Mostefaoui, and Michel Raynal. Preventing useless checkpoints in distributed computations. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, SRDS '97, pages 183–, Washington, DC, USA, 1997. IEEE Computer Society.

[23] R. Baldoni, F. Quaglia, and B. Ciciani. A vp-accordant checkpointing protocol preventing useless checkpoints. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, SRDS '98, pages 61–, Washington, DC, USA, 1998. IEEE Computer Society.

[24] Lorenzo Alvisi, Sriram Rao, Syed Amir Husain, Asanka de Mel, and Elmootazbellah Elnozahy. An analysis of communication-induced checkpointing. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 242–, Washington, DC, USA, 1999. IEEE Computer Society.

[25] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3) :204–226, 1985.

[26] Lorenzo Alvisi and Keith Marzullo. Message logging : Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24 :149–159, February 1998.

[27] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, PODC '89, pages 223–238, New York, NY, USA, 1989. ACM.

[28] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, 1987.

[29] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. On the use and implementation of message logging. In *24th International Symposium on Fault-Tolerant Computing*, pages 298–307. IEEE Computer Society, 1994.

[30] Aurélien Bouteiller, Franck Cappello, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. Mpich-v2 : a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 25–, New York, NY, USA, 2003. ACM.

[31] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and check-pointing. *Journal of Algorithms*, 11 :462–491, September 1990.

[32] How to recover efficiently and asynchronously when optimism fails. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pages 108–, Washington, DC, USA, 1996. IEEE Computer Society.

[33] Sean W. Smith, David B. Johnson, and J. D. Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 361–, Washington, DC, USA, 1995. IEEE Computer Society.

[34] S. L. Peterson and Phil Kearns. Rollback based on vector time. In *12th IEEE Symp. on Reliable Distributed Systems*, SRDS '93, pages 68–77, 1993.

[35] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3 :204–226, August 1985.

[36] S. L. Peterson and Phil Kearns. Rollback based on vector time. In *SRDS*, pages 68–77, 1993.

[37] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho : Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41 :526–531, 1992.

[38] B. Lee, T. Park, H. Y. Yeom, and Y. Cho. An efficient algorithm for causal message logging. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed*

Systems, SRDS '98, pages 19–, Washington, DC, USA, 1998. IEEE Computer Society.

[39] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and orphan-free message logging protocols. In *Proceedings of the 23rd Fault-Toelrant Computing Symposium*, pages 145–154, 1993.

[40] Elmootazbellah Nabil Elnozahy. *Manetho : fault tolerance in distributed systems using rollback-recovery and process replication.* PhD thesis, Houston, TX, USA, 1994. AAI9715028.

[41] Aurelien Bouteiller, Boris Collin, Thomas Herault, Pierre Lemarinier, and Franck Cappello. Impact of event logger on causal message logging protocols for fault tolerant mpi. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, pages 97–, Washington, DC, USA, 2005. IEEE Computer Society.

[42] Thomas Ropars and Christine Morin. Active optimistic message logging for reliable execution of mpi applications. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 615–626, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] Thomas Ropars and Christine Morin. Improving message logging protocols scalability through distributed event logging. In *Proceedings of the 16th international Euro-Par conference on Parallel processing : Part I*, EuroPar'10, pages 511–522, Berlin, Heidelberg, 2010. Springer-Verlag.

[44] Aurelien Bouteiller, Thomas Ropars, George Bosilca, Christine Morin, and Jack Dongarra. Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure recovery. In *IEEE International Conference on Cluster Computing (Cluster 2009)*, New Orleans, USA, 2009.

[45] Sebastien Monnet, Christine Morin, and Ramamurthy Badrinath. Hybrid Checkpointing for Parallel Applications in Cluster Federations. In *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid (CCGRID'04)*, pages 773–782, Washington, DC, USA, 2004. IEEE Computer Society.

[46] Bidyut Gupta, Ruslan Nikolaev, and Raja Chirra. A recovery scheme for cluster federations using sender-based message logging. volume 19, pages 127–139, 2011.

[47] Qi Gao, Wei Huang, Matthew J. Koop, and Dhabaleswar K. Panda. Group-based coordinated checkpointing for mpi : A case study on infiniband. In *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07, pages 47–, Washington, DC, USA, 2007. IEEE Computer Society.

[48] Justin C. Y. Ho, Cho-Li Wang, and Francis C. M. Lau. Scalable Group-Based Checkpoint/Restart for Large-Scale Message-Passing Systems. In *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, USA, 2008.

[49] Aurelien Bouteiller, Thomas Hérault, George Bosilca, and Jack J. Dongarra. Correlated set coordination in fault tolerant message logging protocols. In *Euro-Par (2)*, pages 51–64, 2011.

[50] Nitin Vaidya. Distributed recovery units : An approach for hybrid and adaptive distributed recovery. *Technical Report 93-052, Departement of Computer Science Texas, A&M, University*, 1993.

[51] Jin-Min Yang, Kin Fun Li, Wen-Wei Li, and Da-Fang Zhang. Trading Off Logging Overhead and Coordinating Overhead to Achieve Efficient Rollback Recovery. *Concurrency and Computation : Practice and Experience*, 21 :819–853, April 2009.

[52] Esteban Meneses, Celso L. Mendes, and Laxmikant V. Kale. Team-based Message Logging : Preliminary Results. In *3rd Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids (CCGRID 2010).*, May 2010.

[53] Message P Forum. Mpi : A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[54] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI : Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[55] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22 :789–828, September 1996.

[56] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7) :558–565, 1978.

[57] D. Bailey, T. Harris, W. Saphir, R. van der Wilngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[58] USQCD. http ://www.usqcd.org/usqcd-software/.

[59] Laura Carrington, Dimitri Komatitsch, Michael Laurenzano, Mustafa Tikir, David Michéa, Nicolas Le Goff, Allan Snavely, and Jeroen Tromp. High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62 thousand processor cores. *Proceedings of the ACM/IEEE Supercomputing SC'2008 conference*, pages 1–11, 2008.

[60] Marc Grunberg, Stéphane Genaud, and Catherine Mongenet. Seismic ray-tracing and earth mesh modeling on various parallel architectures. *Journal of Supercomputing*, 29(1) :27–44, 2004.

[61] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research : A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, 2006.

[62] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1) :49–66, 2005.

[63] Jeffrey S. Vetter and Frank Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. *Journal of Parallel and Distributed Computing*, 63 :853–865, September 2003.

[64] Shoaib Kamil, John Shalf, Leonid Oliker, and David Skinner. Understanding ultra-scale application communication requirements. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, pages 178–187, 2005.

[65] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[66] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, 42 :153–159, 1992.

[67] G. Karypis and V. Kumar. *MeTiS : A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0.* Univ. Minnesota, Minneapolis, 1998.

[68] F. Pellegrini. *SCOTCH 5.1 User's Guide*. LaBRI, 2008.

[69] Ü. V. Çatalyürek and C. Aykanat. PaToH : A multilevel hypergraph partitioning tool, version 3.0. Technical Report BU-CE-9915, Bilkent Univ., 1999.

[70] William H. Cunningham. Optimal attack and reinforcement of a network. *J. ACM*, 32 :549–561, July 1985.

[71] John Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *Proceedings of the 2003 international conference on Computational science*, ICCS'03, pages 3–12, Berlin, Heidelberg, 2003. Springer-Verlag.

[72] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17 :530–531, September 1974.

[73] Franck Cappello. Fault tolerance in petascale/ exascale systems : Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23 :212–226, August 2009.

[74] George Bosilca, Aurelien Bouteiller, Thomas Herault, Pierre Lemarinier, and Jack J. Dongarra. Dodging the Cost of Unavoidable Memory Copies in Message Logging Protocols. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 189–197, Berlin, Heidelberg, 2010. Springer-Verlag.

[75] Grid'5000. http ://www.grid5000.fr.

[76] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. NetPIPE : A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[77] BLCR. https ://ftg.lbl.gov/projects/checkpointrestart/.

Table des figures

2.1 État global cohérent 12
2.2 Effet domino 14
2.3 Redémarrage avec un protocole optimiste 17
2.4 Redémarrage avec protocole causal 17

3.1 Émission d'un message 26
3.2 Réception d'un message 26

4.1 Schémas de communication de la primitive *MPI_Alltoall* 40
4.2 Schémas de communication des primitives collectives dans MPICH2 . . . 41
4.3 L'algorithme de partitionnement proposé 43
4.4 Schémas de communications des applications MPI 52

5.1 Déterminisme des émissions 57
5.2 Effet Domino 58
5.3 Scénario d'exécution 59
5.4 Problèmes des causalités 60
5.5 Algorithme du protocole pour les processus de l'application 63
5.6 Algorithme du protocole pour les processus de l'application après une défaillance 64
5.7 Algorithme du procotole pour le processus de redémarrage 65
5.8 Gestion des acquittements pour les messages de petite paille sur un canal de communication 72
5.9 Performances de NetPipe sur Myrinet 10G 74
5.10 Dégradation des performances de NetPipe sur Myrinet 10G 74
5.11 Performances des *NAS Benchmarks* sur Myrinet 10G sur 256 processus . 75
5.12 Utilisation du protocole en définissant des groupes de processus 77
5.13 Performances du protocole sur des groupes de processus 80

6.1 Groupes de processus 85
6.2 Représentation temporelle 85
6.3 Algorithme du protocole pour les processus de l'application en exécution sans faute 86
6.4 Algorithme du protocole pour les processus de l'application en redémarrage 88

6.5 Algorithme du protocole pour les processus de l'application non redémarrés 89
6.6 Algorithme du protocole pour le processus de redémarrage 91
6.7 Algorithme du Processus de Redémarrage distribué 93
6.8 Dégradation des performances de NetPipe sur Myrinet 10G 98
6.9 Performances de Netpipe sur Myrinet 10G 98
6.10 Performances des *NAS Parallel Benchmarks* sur MX sur 256 processus . 99
6.11 Performances des *NAS Parallel Benchmarks* en Redémarrage sur 256 processus 100

Liste des tableaux

3.1 Classification des applications selon les Motifs 28
3.2 Déterminisme des applications MPI 34

4.1 Algorithmes utilisés pour les communications collectives dans MPICH2 40
4.2 Petits groupes 48
4.3 Grands groupes 48

5.1 Résultats avec l'outil de partitionnement 78
5.2 Regroupement de 256 processus 79

6.1 Groupes de processus dans les *NAS Parallel Benchmarks* sur 256 processus 97

7.1 Étude comparative des différents protocoles de tolérance aux fautes . . . 106