Un protocole de recouvrement arrière hiérarchique pour applications MPI de très grande taille avec émissions déterministes

Amina Guermouche, Thomas Ropars

RenPar'20 / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Un protocole de recouvrement arrière hiérarchique pour applications MPI de très grande taille avec émissions déterministes

Amina Guermouche*+, Thomas Ropars+
* Université Paris Sud, F-91405 Orsay, France
+ INRIA Saclay-Île de France, F-91893 Orsay, France

Résumé
Les protocoles de sauvegarde de points de reprise coordonnés sont les protocoles de recouvrement arrière les plus répandus dans les applications MPI de calcul haute performance. Cependant, avec l'augmentation du nombre de composants des machines, les défaillances deviennent de plus en plus fréquentes et le redémarrage de tous les processus de l'application après une défaillance n'est plus une solution adaptée. En s'appuyant sur le déterminisme d'émission de la majeure partie des applications MPI, nous proposons dans cet article un nouveau protocole de recouvrement arrière hiérarchique fondé sur la combinaison d'un protocole de sauvegarde de points de reprise coordonné et d'un protocole à enregistrement de messages. Ce protocole profite des caractéristiques des schémas de communications de la plupart de ces applications qui permettent d'identifier des groupes de processus communiquant fréquemment entre eux, et ainsi d'appliquer un protocole différent au sein des groupes et entre les groupes. Nos évaluations montrent qu'en appliquant un protocole de sauvegarde de points de reprise coordonné au sein des groupes, et en enregistrant seulement les messages entre processus de différents groupes, il est possible de limiter les conséquences d'une défaillance à un petit sous-ensemble des processus de l'application, tout en sauvegardant le plus souvent moins de 20% des données échangées au sein de l'application au cours de son exécution.

Mots-clés : Calcul haute performance, MPI, recouvrement arrière, déterminisme d'émission, protocole hiérarchique

1. Introduction

Avec l'évolution de la taille des machines parallèles, la tolérance aux fautes est devenue une question majeure. Parmi les techniques de tolérance aux fautes existantes, les protocoles de recouvrement arrière fondés sur la sauvegarde de points de reprise sont les plus adaptés pour les applications de calcul haute performance : les images des processus de l'application sont enregistrées périodiquement sur support stable sous forme de points de reprise et récupérées en cas de défaillance, évitant ainsi la perte de tous les calculs déjà effectués.
Dans cet article, nous considérons les applications à échange de messages (MPI). Le type de protocole le plus utilisé pour ces applications sont les protocoles de sauvegarde de points de reprise coordonnés. Ils sont fondés sur la coordination des processus au moment de la sauvegarde des points de reprise afin de garantir un état global cohérent. Lorsqu'une défaillance se produit, tous les processus redémarrent à partir du dernier point de reprise sauvegardé [5]. Ces protocoles ont pour inconvénients : 1) le retour arrière de tous les processus après une défaillance et 2) une possible surcharge du système de fichier liée à l'écriture *simultanée* de tous les points de reprise des processus [6]. D'autres protocoles enregistrent l'ensemble des messages échangés lors de l'exécution afin de réduire le nombre de retours arrière [9]. Cependant, sauvegarder le contenu de tous les messages implique une grande consommation mémoire et induit un surcoût sur les performances des communications [6]. Des solutions alternatives doivent donc être trouvées.
Dans cet article, nous proposons un nouveau protocole de recouvrement arrière hiérarchique exploitant deux caractéristiques que nous avons récemment mises en évidence dans les applications MPI de calcul

haute performance : i) le déterminisme d'émission [7] ; ii) la possibilité d'identifier des groupes de processus communiquant beaucoup entre eux au sein des applications [13]. Notre protocole applique un protocole de sauvegarde de points de reprise coordonné au sein des groupes et un protocole à enregistrement de messages pour les communications entre groupes. Ce protocole permet ainsi 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, 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 rejeu ont été enregistrés ; 2) le déterminisme d'émission assure que l'état des autres groupes restera cohérent avec le groupe redémarrant. Les résultats de nos évaluations avec les *NAS Parallel Benchmarks* [1] montrent que dans le plupart des cas, cette solution permet de limiter à moins de 15% le ratio de processus à redémarrer après une défaillance tout en sauvegardant moins de 20% des messages.
La suite du document est organisée comme suit. Dans le paragraphe suivant, nous présentons le contexte de nos travaux. Le paragraphe 3 décrit les principes et fournit le pseudo-code de notre protocole. Nous présentons nos résultats expérimentaux dans le paragraphe 4. Enfin nous concluons et proposons quelques perspectives dans le paragraphe 5.

2. Contexte

Dans ce paragraphe, nous décrivons d'abord le modèle utilisé dans notre étude. Puis nous revenons sur les principaux protocoles de recouvrement arrière, et sur l'intérêt du déterminisme d'émission. Enfin, nous présentons les protocoles hiérarchiques existants.

2.1. Modèle

Le système considéré est un système distribué asynchrone. 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 fiables et FIFO. Il n'y a pas de délai de transmission des messages et ceux envoyés dans différents canaux ne sont pas ordonnés. Les évènements associés aux émissions et réceptions de messages sont partiellement ordonnés par la relation de précédence causale de Lamport [10]. Les fautes considérées sont des pannes franches et plusieurs fautes peuvent se produire à la fois.

2.2. Le déterminisme dans les protocoles de recouvrement arrière

Un protocole de recouvrement arrière doit garantir qu'un état global cohérent peut être reconstruit après une défaillance. Un état global cohérent est un état qui peut être rencontré dans une exécution sans fautes. Des points de reprise sont sauvegardés tout au long de l'exécution de l'application pour limiter les retours arrière après un défaillance. Lorsqu'un processus redémarre d'un point de reprise, tous les messages qu'il avait envoyés après ce point de reprise ne sont pas contenus dans son état. Cependant, les destinataires de ces messages les ont déjà reçus. De tels messages, reçus mais pas encore envoyés, sont dits "messages orphelins". Ces messages créent un état incohérent entre les processus car il est à priori impossible de garantir que les mêmes messages seront émis lors de la réexécution. Il faut donc que les protocoles de recouvrement arrière gèrent ces messages.
Ces protocoles se divisent en deux grandes familles : les protocoles de sauvegarde de points de reprise et les protocoles à enregistrement de messages. Ces derniers peuvent être associés à un protocole de sauvegarde de points de reprise pour les raisons citées ci-dessus.
Les protocoles de sauvegarde de points de reprise se divisent en trois classes : non coordonné [3], coordonné [5] et induits par les communications [2]. Ils supposent que les applications sont non déterministes, forçant ainsi le retour arrière des processus dépendant d'un message orphelin. La figure 1 représente un exemple d'exécution. Si le processus p_2 subit une défaillance et revient à son dernier point de reprise, rien ne garantit que le message m_1 sera renvoyé, et donc le processus p_1 est forcé de faire un retour arrière pour éviter une incohérence entre les états de p_1 et p_2. Dans la plupart des cas, ceci conduit au retour arrière de tous les processus de l'application après une défaillance. Les protocoles de sauvegarde de points de reprise coordonnés coordonnent les point de reprise sur l'ensemble des processus pour assurer que l'état global sauvegardé soit cohérent. Ils sont le plus souvent employés car ils assurent que le dernier état global sauvegardé est toujours valide limitant ainsi les retours arrières par rapport aux protocoles non coordonnés et simplifiant la suppression des points de reprise inutiles par rapport aux protocoles induits par les communications.

Dans les protocoles à enregistrement de messages, tous les messages sont enregistrés durant l'exécution de l'application pour pouvoir être rejoués après une défaillance. Pour améliorer les performances, le contenu des messages peut être sauvegardé dans la mémoire de l'émetteur [9]. Ils évitent ainsi le retour arrière de tous les processus de l'application après une défaillance. Ils sont fondés sur l'hypothèse du "déterminisme par morceaux" : les seuls évènements non déterministes au cours de l'exécution sont les réceptions de messages. Ainsi, sur la figure 1, si les messages $m_1$et m_2 sont reçus dans cet ordre après la défaillance de p_1, m_3 sera renvoyé. Pour assurer cet ordre de réception, les protocoles à enregistrement de messages stockent, en plus du contenu des messages, l'ordre de leur réception (déterminant) sur support stable. Ainsi, quand p_1 subit une défaillance, il est le seul à revenir à son dernier point de reprise car suffisamment d'informations ont été sauvegardées pour pouvoir rejouer les messages m_1 et m_2 dans le même ordre, assurant donc la réémission de m3, et garantissant que l'état global obtenu sera cohérent. Le principal problème de ce type de solution est qu'enregistrer le contenu de tous les messages de l'application peut induire un surcoût sur les performances des communications et utilise un grand espace de stockage.

2.3. Le déterminisme d'émission

L'étude présentée dans [4] montre que beaucoup d'applications MPI de calcul hautes performance ont des émissions déterministes. Dans ces applications, à partir d'un état initial donné, l'ensemble des processus de l'application émettent toujours les mêmes messages dans le même ordre. Du côté du récepteur, l'ordre de réception des messages non causalement dépendants n'a pas d'effet sur la suite de l'exécution. Sur la figure 1, supposons que le processus p_1 subit une défaillance et revient en arrière. Quel que soit l'ordre de réception des messages m_1 et m_2 lors du rejeu, le message m_3 sera réémis. Le déterminisme d'émission assure donc la réémission des messages orphelins, et permet d'éviter le retour arrière des processus en dépendant, sans avoir à sauvegarder les déterminants des messages. Le paragraphe 3 décrit comment nous exploitons le déterminisme d'émission dans notre protocole. À noter que les applications déterministes sont un sous ensemble des applications à déterminisme d'émission.

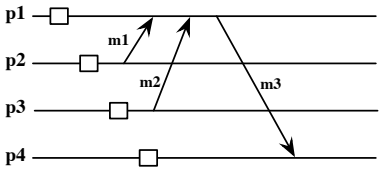

FIGURE 1 – Scénario d'exécution

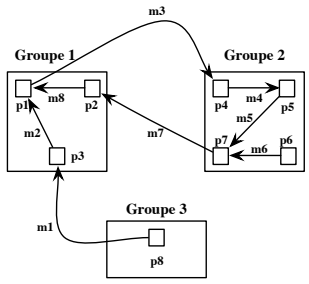

FIGURE 2 – Groupes de processus

2.4. Les protocoles de recouvrement arrière hiérarchiques

Plusieurs travaux proposent de combiner différents protocoles de recouvrement arrière dans un protocole hiérarchique. Certains de ces protocoles sont fondés sur l'aspect hiérarchique des architectures visées [12], d'autres sur les propriétés des schémas de communication des applications. La figure 3 représente le schéma de communication de l'application CG de la suite des *NAS Parallel Benchmarks*. Cette figure illustre le fait que dans de nombreuses applications, il est possible de déterminer des groupes de processus communiquant fréquement entre eux [13]. La figure 4 représente un regroupement possible de ces processus.

Dans cet article, nous exploitons ces groupes de processus pour concevoir un protocole hiérarchique pour applications à émissions déterministes qui utilise un protocole de sauvegarde de points de reprise coordonné au sein des groupes et un protocole à enregistrement de messages entre les groupes. Le but est de limiter les retours arrière en cas de défaillance aux processus du groupe touché par la défaillance tout en n'enregistrant qu'un petit sous ensemble des messages de l'application. Plusieurs travaux ont

présenté des protocoles comparables à celui-ci. Dans [7], nous avons proposé un protocole de sauvegarde de points de reprise non coordonné pour applications à émissions déterministes qui évite l'effet domino (cascade de retours arrières après une défaillance dûs à l'absence d'état global cohérent) en n'enregistrant qu'un sous ensemble des messages échangés dans l'application. Ce protocole permet de définir un ensemble ordonné de p groupes de processus. Les messages allant d'un groupe vers un autre groupe avec un identifiant supérieur sont alors enregistrés, ce qui assure, en cas de défaillance dans un groupe, que seul les groupes avec un identifiant supérieur à ce groupe doivent revenir en arrière. Nous avons démontré que sur un total de p groupes, en moyenne seul $(p+1)/2$ groupes effectuent un retour arrière après la défaillance d'un processus, tout en limitant à moins de 50% le ratio des messages à enregistrer. Le nouveau protocole que nous proposons assure qu'un seul groupe redémarre après une défaillance.

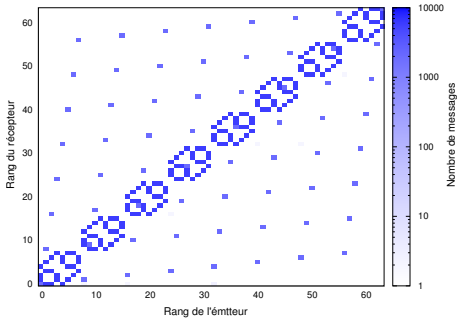

FIGURE 3 – Schéma de communication de CG

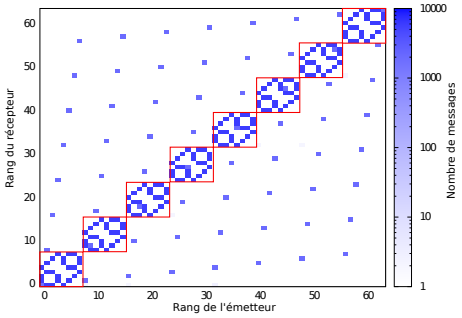

FIGURE 4 – Un regroupement possible

D'autres articles ont proposé des protocoles combinant un protocole de sauvegarde de points de reprise coordonné et un protocole à enregistrement de messages. Certains protocoles [14, 11] sauvegardent les déterminants des messages au sein des groupes, pour pouvoir les rejouer dans le même ordre après un retour arrière et assurer que les messages orphelins seront réémis. Ceci implique des synchronisations avec un support de stockage pour la sauvegarde de ces informations et donc un surcoût sur les performances. Le protocole décrit dans [8] ne sauvegarde aucune information sur les messages échangés au sein des groupes. Dans leur cas, seul un déterminisme total de l'application permet d'assurer que les messages orphelins seront réémis. Nous décrivons dans le paragraphe 3.2 comment nous exploitons le déterminisme d'émission pour assurer la réémission des messages orphelins après une défaillance, sans sauvegarder l'ordre de réception des messages lors de l'exécution normale de l'application.

3. Description

Dans ce paragraphe, nous commençons par décrire notre protocole en fonctionnement normal. Puis, nous décrivons la gestion des défaillances et notamment des causalités. Enfin, nous expliquons comment le ramasse miette est effectué. Les figures 5 et 6 détaillent le pseudo code de notre protocole. Nous utilisons la figure 2, représentant un exemple de regroupement de processus, pour illustrer le fonctionnement de notre protocole. Sur cette figure, les indices des messages représentent l'ordre causal.

3.1. Principes du protocole

Dans ce protocole, le contenu des messages échangés entre les groupes est enregistré au niveau de l'émetteur (lignes 13-14 de la figure 5). Au sein d'un groupe, le protocole utilisé est un protocole coordonné [5]. Ainsi, lorsqu'un processus subit une défaillance, tous les processus du même groupe font un retour arrière (lignes 44-49 de la figure 5). Les processus des autres groupes ne font pas de retour arrière car : 1) les messages destinés aux processus du groupe fautif ont été enregistrés et 2) les messages émis par les processus du groupe fautif et destinés aux autres groupes restent valides grâce au déterminisme d'émission. Sur la figure 2, si un processus du Groupe 2 subit une défaillance causant le retour arrière des processus p_4, p_5, p_6 et p_7, rien ne garantit que l'ordre de réception de m_5 et m_6 sera

Local Variables:

1: P_i {Idenfitiant du processus i}
2: $Groupe_i$ {Identifiant du groupe du processus i}
3: $Date_i \leftarrow 1$, $Phase_i \leftarrow 1$ {Date et phase du processus i}
4: $Redémarrage \leftarrow \emptyset$ {Ensemble des groupes qui sont en redémarrage}
5: $Enregistrès_i \leftarrow \emptyset$ {liste des messages enregistrés par P_i}
6: $RPP_i \leftarrow [\perp, ..., \perp]$ {$RPP_i[j]$ stocke les informations sur les communications entre le processus P_i et le processus P_j appartenant à un autre groupe. $RPP_i[j]$.Maxdate contient la date démission du dernier messages reçus du processus P_j. $RPP_i[j][date]$ contient la phase du message envoyé par P_j à la date $date$}
7: $OrphPhases_i \leftarrow \emptyset$ {Phases dans lesquelles un message orphelin a été reçu par P_i}
8: $RamasseMiette_i \leftarrow [faux, ..., faux]$ {Tableau intialisé à faux permettant de savoir si un message reçu est le premier pour cet émetteur depuis le nouveau point de reprise}
9: $PasEnvoyes \leftarrow \emptyset$ {Stockage temporaire des messages destinés aux processus en cours de redémarrage}
10:
11: **A l'émission du message** msg **au processus** P_j
12: $Date_i \leftarrow Date_i + 1$
13: **si** $Groupe_i \neq Groupe_j$ **alors**
14: $Enregistrés_i \leftarrow Enregistrés_i \cup (P_j, Date, Phase, msg)$
15: **si** $Groupe_j \in Redémarrage$ **alors**
16: $PasEnvoyes \leftarrow PasEnvoyes \cup (msg, Date_i, Phase_i, Groupe_i)$
17: **sinon**
18: Envoyer $(msg, Date_i, Phase_i, Groupe_i)$ au processus P_j
19:
20: **A la réception de** $(msg, Date_{émission}, Phase_{émission}, Groupe_{émission})$ **du processus** P_j
21: **si** $Date_{émission} > RPP_i[j].Maxdate$ **alors** {msg est reçu pour la première fois}
22: **si** $Groupe_{émission} \neq Groupe_i$ **alors**
23: $Phase_i \leftarrow Max(Phase_i, Phase_{émission} + 1)$
24: $RPP_i[j].Maxdate \leftarrow Date_{émission}$
25: $RPP_i[j][date_{émission}].phase \leftarrow Phase_{émission}$
26: **sinon**
27: $Phase_i \leftarrow Max(Phase_i, Phase_{émission})$
28: $Date_i \leftarrow Date_i + 1$
29: **si** $(Groupe_{émission} \neq Groupe_i)$ **et** $(RamasseMiette_i[P_j] = vrai)$ **alors**
30: Envoyer $(Ack, Date_{send})$ à P_j
31: $RamasseMiette_i[P_j] \leftarrow faux$
32: Délivrer msg à l'application
33: **sinon** {le message reçu est un message orphelin}
34: Envoyer $(PlusOrphelinPhase, Phase_{émission})$ au processus de redémarrage
35:
36: **A la sauvegarde du point de reprise**
37: Sauvegarder $(ImagePs_i, RPP_i, Enregistrés_i, Phase_i, Date_i)$ sur support stable
38: $RamasseMiette_i \leftarrow [vrai, ..., vrai]$
39:
40: **A la réception de** $(Ack, MaxDate)$ **de** P_j
41: **pour tout** $date \in Enregistrés_i[P_j]$ **tel que** $date < MaxDate$ **faire**
42: Supprimer $(P_j, date, Phase, Groupe, msg)$ de $Enregistrés_j$
43:
44: **A la défaillance du processus** P_i
45: Récupérer $(ImagePs_i, RPP_i, Enregistrés_i, Phase_i, Date_i)$ du support stable pour le $Groupe_i$
46: **pour tout** $P \in Groupe_i$ **faire**
47: Redémarrer de $ImagePs$
48: $Status \leftarrow Redémarrage$
49: Envoyer $(Redémarrage, Date, Groupe_i)$ à tous les processus de l'application des autres groupes
50:
51: **A la réception de** $(Redémarrage, Date_{re}, Groupe_j)$ **from** P_j
52: $Redémarrage \leftarrow Redémarrage \cup Groupe_j$
53: **pour tout** $(P, Date, Phase, msg) \in Enregistrés_i$ **tel que** $p = P_j$ **et** $Date > Date_{re}$ **faire**
54: Envoyer $(P, Date, Phase, msg, Groupe_i)$ au processus de redémarrage
55: **pour tout** $p_k \in Groupe_j$ **faire**
56: **pour tout** $date \in RPP_i[k]$ **tel que** $date > Date_{re}$ **faire**
57: $OrphPhases_i \leftarrow OrphPhases_i \cup RPP_i[k][date].phase$
58: Envoyer $(Orphelin, OrphPhases_i)$ au processus de redémarrage
59:
60: **A la réception de** $(FinRedémarrage, Groupe_j)$ **du processus de redémarrage**
61: **pour tout** $p \in PasEnvoyer$ **tel que** $p \in Groupe_j$ **faire**
62: Envoyer $(msg, Date_i, Phase_i, Groupe_i)$ au processus p
63: Supprimer $(msg, Date_i, Phase_i, Groupe_j)$ de $PasEnvoyes$
64: Supprimer $Groupe_j$ de $Redémarrage$

FIGURE 5 – Algorithme du protocole pour les processus de l'application

```
Local Variables:
1: NbOrphPhase ← ∅ {NbOrphanPhase[phase] est le nombre de processus qui attendent un message orphelin dans la phase phase}
2: Enregistrés ← ∅ {Enregistrés[phase] est la liste des messages à rejouer envoyés à la phase phase}
3: Groupe_m {Groupe auquel le processus maître appartient}
4:
5: A la réception de (P, Date, Phase, Message, Groupe_j ) du processus P_j
6:     Enregistrés[Phase] ← Enregistrés[Phase] ∪ (P, Date, Message, P_j , Groupe_j )
7:
8: A la réception de (OrphanNotification, OrphPhases_j) du processus P_j
9:     pour tout phase ∈ OrphPhases_j faire
10:        NbOrphPhase[phase] ← NbOrphPhase[phase] + 1
11:    si OrphanNotification a été reçu de tous les processus de l'application alors
12:        Démarrer NotificationPhases
13:
14: A la réception de (PlusOrphelinPhase, Phase) du processus P_j
15:    NbOrphPhase[Phase] ← NbOrphPhase[Phase] − 1
16:    si NbOrphPhase[Phase] = 0 alors
17:        Démarrer NotificationPhases
18:
19: NotificationPhases
20:    pour tout phase ∈ Enregistrés telle que ∄phase' < phase ∧ NbOrphPhase[phase'] > 0 faire
21:        pour tout (P_k , Date, Message, Emetteur, Groupe) ∈ Enregistrés[phase] faire
22:            Envoyer (Message, Date, phase, Emetteur, Groupe) à P_k
23:    si NbOrphPhase = [0, . . . , 0] alors
24:        Envoyer(FinRedémarrage, Groupe_m) à tous les processus
```

FIGURE 6 – Algorithme du processus de redémarrage

la même lors du rejeu. Or, étant donné que ces deux messages ne sont pas causalement dépendants, le déteminisme d'émission assure que quel que soit l'ordre de réception de ces deux messages, le message m_7 sera renvoyé. p_2 n'a donc pas besoin de faire de retour arrière.

3.2. Défaillance et redémarrage

Lorsqu'une défaillance se produit, un processus, appelé "processus de redémarrage", est lancé. Son rôle est de recevoir et d'ordonner en fonction des causalités, les messages sauvegardés à rejouer lors du recouvrement (lignes 5-6 de la figure 6). Si plusieurs groupes subissent des défaillances simultanées, un processus de redémarrage est lancé pour chacun d'eux.

Étant donné que seul le contenu des messages est enregistré, après une défaillance, il faut assurer que l'ordre causal des messages soit respecté afin de garantir la correction de l'exécution. Autrement dit, si l'émission d'un message m' dépend de la réception d'un message m, il faut s'assurer que m' ne sera pas renvoyé avant la réception du message m. Dans une exécution sans faute sur l'exemple de la figure 2, l'émission de m_3 et de m_7 ne pourrait se faire avant la réception de m_1 et m_3 respectivement. En cas de retour arrière des processus du Groupe 1, le message m_3 devient un message orphelin. Le déterminisme d'émission permet de ne pas faire revenir les processus du Groupe 2 afin de recevoir ce message. Ceci pourrait amener m_7 à être renvoyé avant m_1 et donc m_8 à être reçu avant m_2. Or m_8 dépend de m_2, l'exécution serait donc incohérente. Il est clair que la différence avec une exécution sans faute est que les processus du Groupe 2 n'ont plus besoin de recevoir m_3 pour envoyer leurs messages enregistrés. L'existence du message orphelin m_3 est donc la raison de l'incohérence.

Il faut donc que les processus sachent que s'ils reçoivent des messages d'un autre groupe, ces messages peuvent devenir des orphelins, et que les messages qu'ils vont renvoyer en cas de défaillance qui dépendent de ces messages, peuvent mener à un état incohérent.

Afin de garantir que la causalité est respectée, nous adaptons la technique que nous avons proposé dans [7] : utiliser des numéro de phases pour indiquer que les messages dépendent de potentiels orphelins. Ainsi, un message m dépendant d'un autre message m' venant d'un autre groupe doit avoir un numéro de phase supérieur à celui de m'. Lors du redémarrage, le message m' ne pourra être renvoyé qu'à la réception de tous les messages orphelins qui ont une phase inférieure à la sienne. Pour ce faire, à chaque processus est attribuée une phase qui est attachée à chaque message envoyé. Elle est mise à jour au fur et à mesure des réceptions. La phase d'un processus p est mise à jour à la réception d'un message m d'une phase supérieure. Elle est en plus incrémentée de 1 si le message m vient d'un autre groupe

(lignes 22-27 de la figure 5). Sur la figure 2, initialement les phases des processus sont à 1. Lorsque le processus p_3 reçoit le message m_1 (dont le numéro de phase est 1), il incrémente sa phase à 2 et le processus p_1 fait de même à la réception du message m_2. A la réception du message m_3, le processus p_4 met sa phase à 3 puis les processus p_5 et p_7 font de même à la réception des messages m_4 et m_5. Le message m_7 aura donc pour phase 3.
Lorsqu'une défaillance se produit et que le processus de redémarrage récupère tous les messages sauvegardés avec leur numéro phase, il peut commencer à les renvoyer (lignes 11-12 de la figure 6). Afin de pouvoir renvoyer un message qui est à la phase i, il faut s'assurer que tous les messages orphelins dont il dépend ont bien été reçus. Ainsi, chaque fois qu'un processus de l'application reçoit un message orphelin pour une phase, il en informe le processus de redémarrage (lignes 34 de la figure 5). Sur la figure 2, lorsque les processus du $Groupe\ 1$ reviennent en arrière à la suite d'une défaillance, les messages m_1 et m_7 sont renvoyés. La phase de m_1 est 1 alors que celle de m_7 est 3. Étant donné qu'il n'y a pas de messages orphelins à une phase inférieure à 1, m_1 est rejoué mais m_7 ne pourra être rejoué que lorsque p_4 enverra un message au processus de redémarrage pour l'informer qu'il a reçu m_3.
Si un processus non défaillant doit envoyer un nouveau message à un processus en redémarrage, il le sauvegarde dans un tampon et le renvoie une fois le redémarrage terminé (lignes 15-16 et 60-64 de la figure 5).

3.3. Suppression des données obsolètes

Dans un protocole de sauvegarde de points de reprise coordonné, 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 message enregistrés par les processus des autres groupes, et reçus avant ce point de reprise, ne sont plus nécessaires. Pour pouvoir supprimer ces message, aprés avoir pris un point de reprise, chaque processus répond par un acquitement au premier message de chaque processus des autres groupes (lignes 29-31 de la figure 5), permettant au processus recevant l'acquitement de supprimer les messages destinés à ce processus dont la date est antérieure à celle contenue dans l'acquitement (ligne 40-42 de la figure 5).

4. Evaluation

L'ensemble des résultats présentés dans ce paragraphe ont été obtenus en exécutant les applications sur la grappe de Rennes de la plateforme Grid'5000. Les applications utilisées sont BT, CG, LU, MG, SP et FT de la suite des NAS Parallel Benchmark, en taille "class D" et exécutées sur 1024 processus. Nous avons exécuté les applications avec une version modifiée de MPICH2 afin d'intercepter toutes les communications au sein de l'application. A partir de ces données, nous avons employé l'algorithme de partitionnement proposé dans [13] pour obtenir un partitionnement adapté à notre algorithme. Il fonctionne comme suit : Chaque application est exécutée une première fois afin de récupérer l'ensemble des informations sur les communications entre les processus. À partir de ces données, l'algorithme divise l'ensemble des processus en groupes par bisections successives, en exploitant une fonction de coût pour évaluer chaque partitionnement.
Le tableau 1 présente les résultats obtenus à partir de cet outil de partitionnement. Pour chaque application, la table contient le nombre de groupes créés par l'outil, la taille maximale et minimale des groupes, le pourcentage de processus qui font un retour arrière en moyenne en cas de défaillance et le pourcentage de messages enregistrés en terme de quantité de données.
Les résultats montrent que, dans la plupart des cas, il est possible de créer des groupes dont la taille moyenne est inférieure à 15% de la taille de l'application, tout en ayant moins de 20% de messages enregistrés. Néanmoins, FT montre un taux assez élevé de processus à redémarrer et de messages enregistrés. FT utilise beaucoup de communications de type *all-to-all*, c'est-à-dire où tous les processes de l'application communiquent entre eux. Ceci complique le partitionnement, mais ne peut de toute façon être utilisé que de façon limitée à très large échelle.

5. Conclusion

Dans cet article, nous avons proposé un protocole de recouvrement arrière hiérarchique pour les applications MPI de calcul haute performance à émissions déterministes. Le protocole partitionne les processus en groupes, exploitant l'existence de groupe de processus communiquant fréquemment entre eux

Applications	Nombre de groupes	Taille Min	Taille Max	% Retour	% enregistré
BT	8	123	132	12.50%	12.33%
CG	32	32	32	3.125%	16.23%
LU	16	64	64	6.25%	9.68%
MG	8	128	128	12.5%	18.43%
SP	8	124	133	12.5%	12.14%
FT	2	502	522	50%	50%

TABLE 1 – Pourcentage des processus en redémarrage et de messages enregistrés

au cours de l'exécution. Notre protocole combine sauvegarde de points de reprise coordonné au sein des groupes et enregistrement de messages entre les groupes. Il permet ainsi de limiter, d'une part, le nombre de processus qui font un retour arrière après une défaillance, et, d'autres part, la taille totale des messages enregistrés au cours de l'exécution. De plus, le protocole proposé exploite le déterminisme d'émission pour éviter la sauvegarde de l'ordre de réception des messages au sein des groupes et ainsi limiter les surcoût en terme de performance en fonctionnement normal. Nos évaluations montrent que le protocole atteint les objectifs souhaités pour la plupart des applications : 1) moins de 15% des processus font un retour arrière après une faute et 2) moins de 20% des messages sont enregistrés.

Nous avons commencé la mise en œuvre du protocole dans MPICH2, en intégrant notamment la notion de groupes au gestionaire de processus, pour pouvoir évaluer expérimentalement son intérêt en terme de perfomances, mais aussi en terme d'énergie consommée pour la gestion des défaillances.

Bibliographie

1. 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.
2. R. Baldoni. A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, FTCS '97, pages 68–, Washington, DC, USA, 1997. IEEE Computer Society.
3. B. Bhargava and S.-R. Lian. Independent Checkpointing and Concurrent Rollback for Recovery in Distributed Systems - an Optimistic Approach. pages 3–12, 1988.
4. F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *19th International Conference on Computer Communications and Networks (ICCCN 2010)*, pages 1–8, 2010.
5. K. Chandy and L. Lamport. Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1) :63–75, 1985.
6. E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3) :375–408, 2002.
7. A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic Message Passing Applications. In *25th IEEE International Parallel & Distributed Processing Symposium (IPDPS2011)*, Anchorage, USA, 2011.
8. J. C. Y. Ho, C.-L. Wang, and F. 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.
9. D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *In Digest of Papers : 17 Annual International Symposium on Fault-Tolerant Computing*, pages 14–19. IEEE Computer Society, 1987.
10. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7) :558–565, 1978.
11. E. Meneses, C. L. Mendes, and L. 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.
12. S. Monnet, C. Morin, and R. 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.
13. T. Ropars, A. Guermouche, B. Uçar, E. Meneses, L. V. Kalé, and F. Cappello. On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications. Technical Report TR-JLPC-11-01, INRIA-UIUC Petascale Computing Jointlab, 2011.
14. J.-M. Yang, K. F. Li, W.-W. Li, and D.-F. Zhang. Trading Off Logging Overhead and Coordinating Overhead to Achieve Efficient Rollback Recovery. *Concurrency and Computation : Practice and Experience*, 21 :819–853, April 2009.