ANNEE 2005/2006

RAPPORT de «MISSION en ENTREPRISE »

présenté par

Camille Coti

mission effectuée du 1/04/06 au 15/09/06 chez :
INRIA Futurs

sujet de la mission :

Conception et évaluation d'un algorithme de tolérance aux fautes à points de reprises coordonnées pour MPICH2

Directeur de stage : **Thomas Hérault**
Conseiller de stage : **Éric Renault**

Travail effectué pour la société :
INRIA Futurs
addresse de l'INRIA

Stage de fin d'études
Conception et évaluation d'un algorithme de tolérance aux fautes à points de reprises coordonnées pour MPICH2

Camille Coti

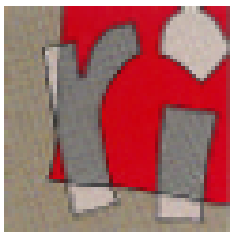
1er avril - 15 septembre 2006

Abstract

This traineeship took place in the Grand Large team within INRIA Futurs, located in the Laboratoire de Recherche en Informatique of the Orsay University. I worked on a project called MPICH-V whose goal is to design, assess and implement fault-tolerance protocols for large-scale message-passing applications. As part of this project, I first contributed to the implementation of a fault-tolerance protocol based on the Chandy-Lamport algorithm. In particular I developped one component, and designed, implemented and evaluated a hierarchical fault-tolerance protocol.

Résumé

J'ai effectué ce stage au sein de l'équipe Grand Large de l'INRIA Futurs, installée au Laboratoire de Recherche en Informatique de l'Université d'Orsay. Le projet auquel j'ai été intégrée, MPICH-V, a pour but de concevoir, d'évaluer et d'implémenter des protocoles de tolérance aux pannes dans les applications parallèles par passage de messages, en particulier dans des systèmes à grande échelle. C'est dans ce cadre que j'ai été amenée dans un premier temps à participer à l'implémentation d'un protocole de tolérance aux pannes basé sur l'algorithme de Chandy-Lamport en en développant un composant, puis à concevoir un protocole hiérarchique de tolérance aux pannes et à l'évaluer en réalisant une implémentation prototypaire.



Remerciements

Je remercie toutes les personnes du Laboratoire de Recherche en Informatique et les membes de l'équipe Grand Large qui ont facilité mon intégration au sein de l'équipe, par ordre alphabétique : Fatiha Bouabache, Gilles Fedak (qui est un peu à l'origine de tout ça), Gina Grivsard, William Hoarau,Derrick Kondo, Julien Leduc, Paul Malecot, Vincent Neri, Laurence Pilard, Benjamin Quétier, Ala Rezmerita, Éric Rodriguez, David Soguet et Sébastien Tixeuil.

Je tiens à remercier particulièrement Thomas Hérault pour m'avoir encadrée durant ce stage, pour sa patience, sa pédagogie et sa disponibilité, Pierre Lemarinier qui a participté à l'encadrement de ce stage, et Franck Cappello pour m'avoir acceptée dans son équipe.

Enfin, je remercie Éric Renault qui a accepté d'être mon conseiller de stage, et Guy Bernard.

Table des matières

1 Introduction **9**
1.1 Pourquoi tolérer les défaillances dans les applications MPI . . 9
1.2 Sujet 10
1.3 Présentation du laboratoire 10
1.3.1 Le LRI 10
1.3.2 INRIA et INRIA Futurs 12
1.3.3 Le projet Grand Large 12
1.4 Grid'5000 12
1.5 Le projet MPI-V 14

2 Protocoles de tolérance aux pannes **15**
2.1 Définitions 15
2.2 Types de fautes 15
2.3 Redondance 16
2.4 Retour sur points de reprise 18
2.4.1 Reprise sur points coordonnés 20
2.4.2 Message logging 23
2.4.3 CIC 28

3 MPI et ses implémentations **30**
3.1 Message-Passing Interface 30
3.1.1 MPI-1 30
3.1.2 MPI-2 30
3.2 MPICH 31
3.2.1 MPICH1 31
3.2.2 MPICH2 31
3.3 LAM/MPI 31
3.4 Open MPI 31
3.5 Autres implémentations 32

4 MPICH2-*pcl* **34**
4.1 Algorithme 34
4.2 Architecture et implémentation 34
4.2.1 Prise des checkpoints 34
4.2.2 Architecture du système 35
4.2.3 Implémentation dans MPICH2 35
4.3 Le serveur de checkpoints 36
4.4 Comparaison *pcl* vs *vcl* 40
4.4.1 MPICH-*vcl* 40

4.4.2 Expérimentations 40

5 MPICH-*Vcausal* pour grilles **48**
5.1 Problématique 48
5.2 Travaux antérieurs 48
5.3 Protocole 49
5.3.1 La Piecewise Deterministic assumption sur une grille . 49
5.3.2 Algorithmes 51
5.4 Implémentation 54
5.5 Performances 56
5.5.1 Démarche expérimentale 56
5.5.2 Conditions d'expérimentations 60
5.5.3 Résultats et interprétation 60

6 Conclusion **64**

1 Introduction

1.1 Pourquoi tolérer les défaillances dans les applications MPI

Le domaine du calcul à hautes performances mutualise aujourd'hui de plus en plus de ressources de calcul collaborant à une tâche commune. On assiste à une augmentation du nombre de processeurs dans les clusters, qui sont eux-même connectés pour former des grilles. Les super-calculateurs recensés dans le top500 de juin 2006 [1] comptent pour la plupart plus de 1000 processeurs par machine, allant jusqu'à 131 072.

Même avec les composants les plus fiables, des pannes risquent de survenir dans de tels systèmes. Il est évident que si l'on augmente le nombre de composants dans un système, la probabilité qu'une panne survienne dans ce système augmente. Une machine de l'ensemble sur lequel l'application s'exécute est alors perdue pour ce calcul. De plus, les clusters d'une grille sont interconnectés par un réseau de type Internet, non maîtrisé par l'administrateur de la grille et sujet aux pannes : il est alors possible de perdre un cluster entier. Enfin, les connexions entre les clusters constituent des points centraux de défaillances en passant par des machines frontales (qui peuvent tomber en panne) et en traversant parfois plusieurs réseaux.

Or, une panne dans un composant du système, si elle n'est pas tolérée, fait perdre le calcul exécuté par le système dans son ensemble. Les calculs exécutés sur de tels systèmes étant souvent destinés à durer plusieurs heures voir plusieurs jours, la probabilité qu'une panne survienne pendant le calcul est forte et la perte engendrée importante. Dans une application parallèle, chaque processeur collabore au calcul de manière à ce que celui-ci s'exécute plus rapidement qui s'il était exécuté de manière séquentielle sur une seule machine. Si une panne survient sur une machine faisant partie d'un système composé de 500 processeurs au bout d'une journée de calculs, la quantité de calcul perdue est équivalente à celle produite en un an par un seul processeur exécutant une application séquentielle.

Il est donc dans ces conditions indispensable de pouvoir tolérer les fautes de manière à ce que le calcul puisse continuer à s'éxécuter et se terminer malgré les pannes, le tout en conservant des performances maximales (objectif de hautes performances).

Parallèlement à l'explosion du nombre de processeurs impliqués dans les calculs à hautes performances, le paradigme de programmation d'applications parallèles par passage de messages (ou MPI) s'est imposé comme le standard *de facto* pour la programmation d'applications de calcul à hautes performances s'exécutant sur des systèmes multi-processeurs à mémoire distribuée.

Le but est ici de tolérer les pannes dans les applications parallèles par passage de messages de manière transparente. Cette transparence s'exprime pour l'utilisateur par le fait qu'il ne doit pas s'apercevoir qu'une panne est survenue dans le système pendant qu'il exécute une application, et pour le programmeur, en ne modifiant pas de contraintes de programmation différentes dans le but qu'un code écrit pour un environnement non tolérant aux fautes puisse être exécuté dans un environnement tolérant aux fautes sans nécessiter de modifications, et en ne le faisant pas intervenir dans les mécanismes de tolérances aux pannes.

1.2 Sujet

J'ai travaillé durant ce stage au sein du projet MPI-V, qui a pour but d'implémenter à des fin d'évaluation des protocoles de tolérance aux fautes dans les applications parallèles par passage de messages. Plus précisément, j'ai participé au projet MPICH2-V à travers l'implémentation d'un mécanisme de toléranceaux pannes pour MPICH2.

Dans un premier temps, j'ai participé à l'implémentation d'un protocole de recouvrement d'exécution par retour sur points de reprise coordonnés basé sur l'algorithme de Chandy-Lamport en prenant part au développement d'un de ses composants.

La deuxième partie de ce stage a consisté à la conception et l'étude d'un autre protocole de tolérance aux pannes adapté aux grilles de calcul et basé sur l'enregistrement des messages échangés dans l'application.

1.3 Présentation du laboratoire

1.3.1 Le Laboratoire de Recherche en Informatique

Le Laboratoire de Recherche en Informatique est, avec le LIMSI, une des deux unités de recherche en informatique de l'Université Paris XI. C'est une

unité de recherche mixte Université Paris XI et CNRS. Il est constitué de 10 équipes de recherche :

- Algorithmique et Complexité
- Architectures parallèles
- Bases de Données
- Bioinformatique
- Démonstration et programmation
- Inférence et Apprentissage
- Intelligence Artificielle et systèmes d'inférence
- Parallélisme
- Programmation - Génie logiciel
- Théorie des Graphes et Fondements des Communications

J'ai effectué ce stage dans l'équipe *Parallélisme*, dirigée par Brigitte Rozoy et formée de 8 membres permanents, 2 attachés temporaires d'enseignement et de recherche et 4 doctorants. La thématique principale de l'équipe est les algorithmes répartis tolérants aux pannes, déclinés sous deux axes de recherche : les algorithmes répartis et le calcul à grande échelle.

Le premier axes concerne les algorithmes répartis, en particulier les algorithmes auto-stabilisants. Un algorithme auto-stabilisant est un algorithme dont les variables ne sont pas initialisées et qui doit fonctionner quelle que soit l'initialisation. Cette absence d'initialisation permet de modéliser n'importe quelle défaillance. Un algorithme auto-stabilisant doit vérifier sa spécification au bout d'un temps fini, quelle que soit l'initialisation. Les travaux de recherche concerne la modelisation de systèmes, la conception d'algoritthmes et l'étude de leur complexité.

Le deuxième axe concerne le calcul à grande échelle à travers la modélisation de systèmes, la conception de protocoles et l'analse de performances. Il s'agit d'une autre approche de la tolérance aux pannes que l'autostabilisation, ici par retour arrière. La tolérance aux pannes est une nécessité pour les raisons citées ci-dessus afin de pouvoir faire des calculs nécessitant beaucoup de puissance de calcul sur des clusters ou des grilles de grande taille. Les problèmes à résoudre sont l'écriture de ces applications (techniques logicielles parallèles), et comment pouvoir tolérer les défaillances pendant l'exécution. L'équipe parallélisme collabore sur ce travail avec le projet Grand Large de l'INRIA Futurs, dont fait également partie l'équipe architecture parallèle.

1.3.2 INRIA et INRIA Futurs

L'INRIA est l'Institut National de Recherche en Informatique et en Automatique. C'est un organisme public de recherche qui finance des projets de recherche au sein d'unités de recherche. Les unités de recherche sont constituées de chercheurs INRIA et de chercheurs provenant de laboratoires partenaires.

L' INRIA Futurs est une unité de recherche de l'INRIA créée en janvier 2002 regroupant des projets de recherche en partenariat avec 31 équipes dans des laboratoires existants :

- le LRI (CNRS et Université de Paris-Sud)
- le LIX (CNRS et Ecole Polytechnique)
- l' ENSEIRB et Université de Bordeaux I
- le LABRI et le MAB (CNRS, Universités de Bordeaux I et Bordeaux 2, ENSEIRB)
- le LIFL (CNRS et Université des sciences et technologies de Lille)
- le LSV (CNRS et ENS Cachan)

L'effectif de l'INRIA Futurs compte environ 300 personnes (INRIA : 3600).

L'INRIA Futurs finance, entre autres, le projet Grand Large.

1.3.3 Le projet Grand Large

Le projet Grand Large est un projet commun entre l'INRIA Futurs, le LRI et le LIFL (à Lille). Dirigé par Franck Cappello, Directeur de Recherche à l'INRIA et apparenté à l'équipe Architectures Parallèles, ce projet regroupe au LRI des membres des équipes "Théorie des Graphes et Fondements des Communications", "Parallélisme" et "Architectures Parallèles" ainsi que des chercheurs permanents de l'INRIA. Il a pour but d'étudier le calcul dans les systèmes à grande échelle comme dans les grilles ou dans les grands systèmes pair-à-pair en situant son approche au niveau middleware.

1.4 Grid'5000

FIG. 1 – Grid'5000

Grid'5000 [12] est une plate-forme expérimentale dédiée à la recherche dans le domaine du calcul sur grille à grande échelle. C'est une plate-forme destinée uniquement à la recherche en informatique et non pas à la production.

Elle est contituée de clusters situés dans 17 laboratoires de recherche distribués sur le territoire Français dans 9 villes et reliés par le réseau Renater[1].

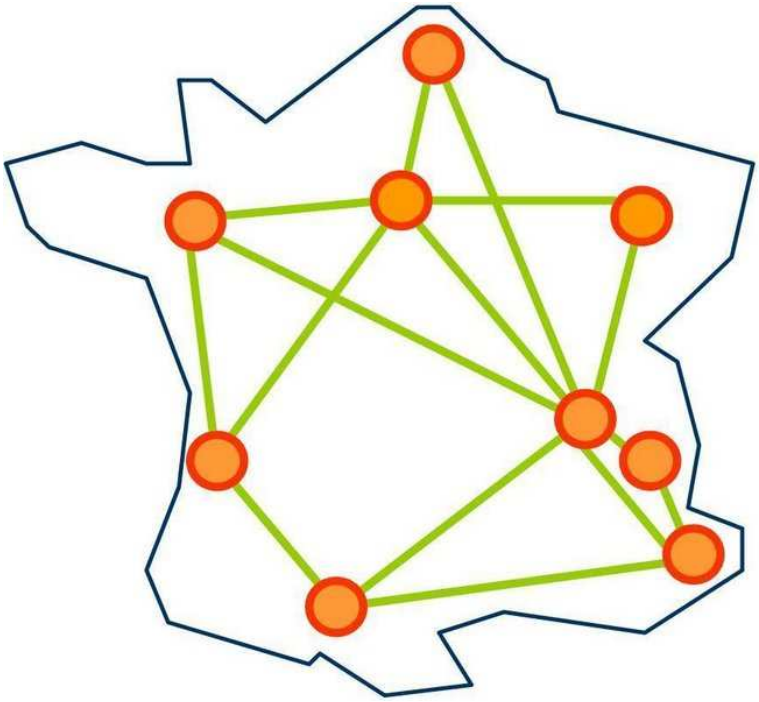


FIG. 2 – Interconnexction des clusters de Grid'5000

Quatre types d'architecture sont disponibles : Opteron, Xeon, G5 et Itanium. La quantité de mémoire disponible pour chaque processeur diffère selon les clusters, de même que les réseaux utilisés. Les plans de partitionnement des disques durs doivent être le même sur toutes les machines pour des raisons liées aux outils de déploiement.

En effet, une spécificité de Grid'5000 est sa flexibilité due au fait que chaque utilisateur a la possibilité, au démarrage, d'installer son propre environnement sur les machines dont il dispose. Il peut alors utiliser le système d'exploitation de son choix, avec les versions des bibliothèques et des logiciels dont il a besoin.

[1]Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis, Toulouse

L'outil de réservation, Oar, couplé à celui de déploiement, Kadeploy, permet à l'utilisateur de disposer de machines qui lui sont dédiées exclusivement sur un temps fixé au moment de la réservation, et d'être administrateur sur ces machines.

1.5 Le projet MPI-V

Le projet MPI-V a pour but d'étudier les mécanismes de tolérance aux pannes dans les applications MPI. Les protocoles existants et connus en théorie sont implémentés, et de nouveaux protocoles sont créés. Sont évalués leur influence sur les performances de l'application en l'absence de fautes, et le degré de pénalisation des fautes en cours d'exécution.

Il s'agit d'un sous-projet du projet Grand Large ; ces protocoles sont évalués dans une optique de grande échelle. Après avoir exploré les protocoles majeurs dans des clusters, le projet se tourne aujourd'hui davantage vers les grilles, où moins de travaux ont été menés jusqu'à présent.

Ce projet est mené en collaboration avec les équipes de MPICH à Argonne National Laboratories pour l'implémentation dans MPICH (MPICH-V et MPICH2-V) et l'University of Tennessee, Knoxville pour l'implémentation dans OpenMPI.

2 Protocoles de tolérance aux pannes

2.1 Définitions

Les systèmes distribués sont difficiles à définir de manière unique. On dit qu'il existe autant de définitions qu'il existe de systèmes distribués. Andrew Tanenbaum les définit comme un ensemble d'ordinateurs indépendants qui apparaissent aux utilisateurs comme un seul système cohérent. Leslie Lamport définit un système distribué comme un système dans lequel vous ne pouvez pas travailler parce qu'un ordinateur dont vous n'avez jamais entendu parler ne fonctionne plus. Plus généralement, on peut dire qu'un système distribué est un ensemble d'entités autonomes de calcul interconnectées et qui peuvent communiquer.

Un processus est une entité représentant une activité déterminée par un programme qui est exécutée sur un processeur. À chaque processus sont associées :

- un programme qui détermine ce que doit faire le processus
- une entrée et une sortie
- un état représenté par la valeur des variables du programme

Un canal de communications est un médium utilisé pour transmettre des informations entre un émetteur et un receveur.

On traite ici de systèmes distribués constitués de plusieurs processus communiquant entre eux *via* des canaux de communications.

2.2 Types de fautes

Il existe plusieurs types de fautes, classables en deux grandes catégories : temporaires ou définitives.

Les pannes temporaires peuvent être de nature différente. On parle d'omission lorsqu'un processus ou un canal ne réalise pas une action. À l'inverse, la duplication correspond au fait qu'un canal ou un processus réalise deux fois une action qui n'aurait dû se produire qu'une seule fois. Enfin, une inversion correspond à la réalisation d'une action B avant une action A, alors que A aurait dû avoir lieu avant B.

Les fautes byzantines peuvent être temporaires ou définitives. Un processus ou un canal a un comportement arbitraire erroné, mais la détection des erreurs ne se fait pas aussi facilement que dans les cas dans autres types de fautes. Ainsi, l'entité fautive peut interagir avec les autres entités du système, et la faute se propage dans le système.

Enfin, les pannes franches sont des pannes définitives : un processus ou un canal ne réalise plus aucune action. On parle alors de silence sur défaillance. Ces pannes sont des omissions permanentes. On s'intéresse ici à ce type de pannes, appelées aussi *pannes crash* ou *fail stop*.

Une panne crash peut correspondre à une défaillance matérielle : panne franche du processeur, coupure physique (électricité, réseau)... Il existe actuellement des travaux portant sur la prévention de ces pannes, pour faire migrer les processus d'une machine allant tomber en panne vers une machine stable actuellement, afin d'éviter la panne. Ils se basent par exemple sur la mesure de la température, car il a été remarqué qu'une augmentation de la température au-delà d'un certain seuil engendre une surchauffe et une panne. Une autre vision plus ancienne de la tolérance aux défaillances implique de détecter la panne une fois qu'elle a eu lieu, et la gérer par une méthode logicielle.

La redondance est une solution ancienne et qui n'est plus étudiée dans le domaine d'applications présent. Cependant, pour certaines utilisations des grilles, on l'utilise encore. L'attention ici se porte sur des protocoles de reprise sur points, coordonnés ou non. S'il existe plusieurs catégories de protocoles, aucun ne sort vraiment du lot. Une revue de ces protocoles et des évaluations qui en ont été réalisées figure dans [17]. On peut aussi trouver un aperçu des implémentations existantes dans [23].

2.3 Redondance

La première solution proposée fut la redondance. Pour tolérer f fautes, il faut alors disposer de tous les composants du système f fois. La tolérance aux fautes divise donc les capacités (calcul, stockage) du système par f.

La réplication peut cependant s'avérer utile dans des applications de grille où l'objectif est d'avoir le résultat du calcul rapidement mais sans avoir besoin de toutes les ressources matérielles de la grille. On n'a alors pas la possibilité de ralentir le calcul en effectuant des retours en arrière en cas de panne.

Il existe deux techniques majeures de réplication, décrites dans [25] : la réplication active, et la réplication primaire/sauvegarde.

La réplication primaire/sauvegarde est transparente vis-à-vis des autres composants du système qui ne sont en contact qu'avec un seul réplicat dit primaire. C'est ce réplicat primaire qui effectue la réplication auprès des réplicats secondaires (pour la sauvegarde, ou backups), et attend d'avoir reçu le résultat renvoyé par chacun des réplicats secondaires et d'avoir lui-même effectué l'action pour considérer l'action comme effectuée et renvoyer le résultat (cf figure 3). Dans le cas, par exemple, d'un calcul à effectuer, chaque réplicat effectue ce calcul (en rouge sur la figure 3). Chaque réplicat secondaire envoie son résultat au réplicat primaire, qui ne renvoie le résultat qu'une fois tous les calculs effectués.

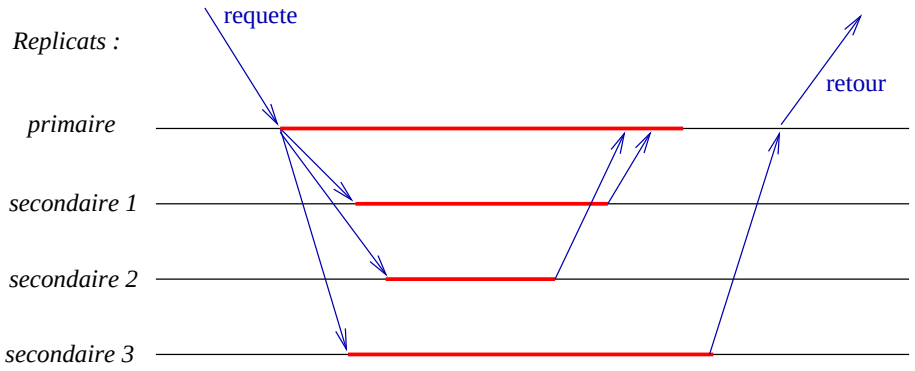


FIG. 3 – Mécanisme de réplication passive avec 3 réplicats secondaires

Au contraire de la réplication primaire/sauvegarde, la réplication active n'est pas transparente vis-à-vis du reste du système. Elle est également appelée approche par machine à états [33]. Tous les réplicats ont un rôle similaire, et c'est à l'appelant du calcul d'invoquer chacun des réplicats, qui lui retourneront directement le résultat du calcul. Cette technique est illustrée figure 4.

La réplication primaire/sauvegarde présente l'inconvénient d'introduire une latence importante lors de la réplication : en effet, le résultat n'est renvoyé qu'une fois que tous les réplicats de sauvegarde ont effectué l'action, donc pas avant que le plus lent des réplicats ne l'ait effectuée. De plus, elle nécessite des communications supplémentaires. La réplication active permet d'éviter

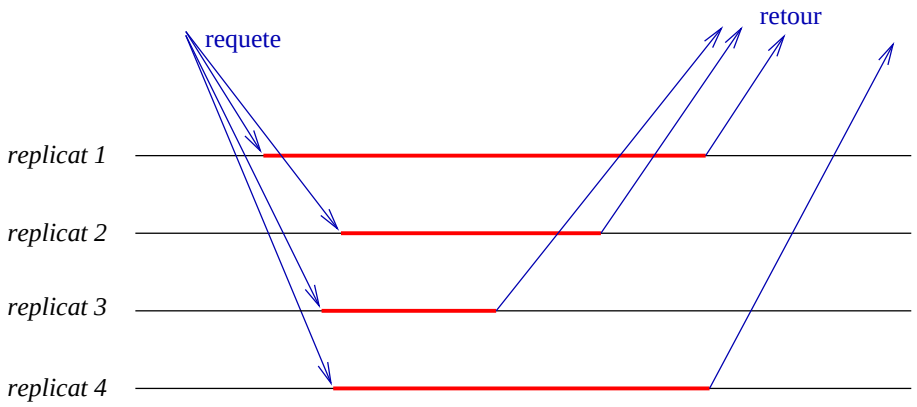


FIG. 4 – Mécanisme de réplication active avec 4 réplicats

cette latence, cependant son absence de transparence peut être un inconvénient. En effet, le reste du système a alors besoin d'avoir une connaissance de l'ensemble de réplicats disponibles et de leur localisation.

Il a été proposé des variantes de ces techniques dans [39]. Cet article se place dans un objectif de temps de réponse : la réplication a ici pour but d'obtenir un temps de réponse minimal, avant d'assurer la tolérance aux pannes. Plutôt que d'attendre la réponse de tous les réplicats secondaires dans le cas de la réplication passive, le réplicat primaire attend la réponse d'un quorum de réplicats secondaires, ce qui diminue l'attente engendrée par les réplicats les plus lents.

2.4 Retour sur points de reprise

Les points de reprise (checkpoints)

L'exécution d'un processus est une succession d'actions qui le font passer d'un état à un autre. La reprise sur point consiste à faire reprendre son exécution à un processus à partir d'un état précédent sauvegardé lors d'une exécution précédente. On sauvegarde les états selon une heuristique définie par l'algorithme du protocole utilisé. L'enregistrement de l'état d'un processus à un instant donné est appelé *checkpoint*. L'ensemble des checkpoints d'un système distribué est un *snapshot*.

Les protocoles de retour sur points de reprise impliquent que les différents snapshots soient sauvegardés sur un support de stockage que l'on considère comme stable afin de ne pas être concernés par une panne. Un processus situé

sur une machine tombée en panne peut être migré sur une machine saine et reprendre son exécution à partir d'un checkpoint précédemment sauvegardé.

État cohérent et ligne de recouvrement

Pour pouvoir reprendre le calcul après une panne, il faut que tous les processus impliqués dans celui-ci repartent d'un *état cohérent*, c'est-à-dire d'un état dans lequel le système aurait pu être au cours du calcul et ne modifiant pas le résultat du calcul si on revient dessus. Un état est dit *cohérent* si pour tous les messages m d'un processus P_i vers P_j, si le checkpoint sur P_j a été effectué après la réception de m alors le checkpoint de P_i a été effectué après l'émission de m [29]. Il est impératif que le fait qu'une panne soit survenue n'ait pas d'influence sur le résultat du calcul en cours. Or, une application parallèle par passage de message est une application distribuée, et à un instant donné, on ne sait pas si un message est en train de transiter sur le réseau. Il faut donc tenir compte des messages qui circulent, et de l'absence d'horloge globale entre les processus.

On peut voir figure 5 un exemple de ligne de recouvrement constituant un état incohérent. On voir en rose le moment où les processus effectuent leur checkpoint, et en bleu la ligne de recouvrement constituée. On constate deux messages (en vert) traversant cette ligne de recouvrement : le message m1 a été envoyé après que le processus 1 ait pris sont checkpoint et reçu avant que le processus 2 n'ait pris le sien. En cas de retour en arrière des processus sur cette ligne de recouvrement, le processus 2 considèrera le message 1 comme reçu alors que le processus 1 considèrera ne pas l'avoir envoyé. De même, le message m2 traverse cette ligne en étant envoyé en aval et reçu en amont. En cas de retour en arrière, le procesus 4 attendra la réception de m2, alors que le processus 3 considèrera l'avoir déjà envoyé.

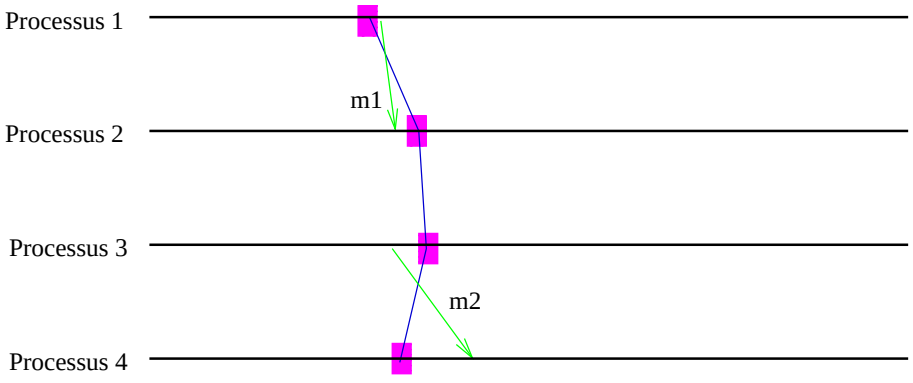


FIG. 5 – Ligne de recouvrement traversée par des messages

Lorsque le système effectue un retour en arrière, il reprend son exécution dans le dernier état cohérent sauvegardé : on parle alors de *ligne de recouvrement*. Il faut évidemment que cette ligne de recouvrement soit la plus avancée possible, afin de perdre le moins possible de temps lors du retour en arrière et diminuer le coût de cette faute. S'il est impossible actuellement de ne pas subir de surcoût lorsqu'une panne survient, la recherche dans ce domaine cherche à le diminuer le plus possible mais les fautes engendrent pour le moment inévitablement un surcoût en matière de temps d'exécution. Les protocoles de checkpoints coordonnés font repartir l'ensemble du système du dernier état enregistré : tous les processus effectuent un retour en arrière. Les protocoles de checkpoint non coordonnés ne font revenir en arrière que le processus concerné par la panne, et peuvent faire revenir en arrière les processus dont l'exécution dépend causalement du processus qui est retourné en arrière : par exemple pour faire envoyer un message dont la réception s'est faite après la dernière prise de checkpoint. La propagation des retours en arrière des processus peut aller jusqu'à reprendre le calcul à partir du début, comportement appelé *effet domino* [31]. La tolérance aux pannes ne sert alors à rien, si ce n'est à ralentir le calcul. L'enregistrement des messages échangés dans le but de pouvoir les rejouer en cas de retour en arrière permet d'éviter cet effet domino.

Dans tous les protocoles présentés ci-après, on suppose que les canaux de communication ont la propriété FIFO. En pratique, cette hypothèse n'est pas très forte, on se permet donc de la faire. De plus, les bibliothèques de passage de messages sont implémentées sur le protocole TCP, qui remet les paquets dans l'ordre au moment de la réception.

2.4.1 Reprise sur points coordonnés

L'algorithme de Chandy-Lamport est un exemple de prise de checkpoint de tout le système dans un état cohérent avec l'introduction de l'idée de snapshot distribué [13]. Un processus du système a un rôle d'initateur du checkpoint, en faisant circuler un marqueur. Son implémentation peut se faire en envoyant ce marqueur à tous les processus, qui doivent alors vider leurs canaux de communication et en sauvegarder l'état, ou en diffusant ce marqueur par vague : les processus ayant reçu le marqueur le transmettent à leur tour.

Checkpoints bloquants ou non bloquants

Les protocoles de checkpoints bloquants arrêtent leur exécution pendant la prise du checkpoint et ne la reprennent que lorsque tous les processus l'ont

fait. Ainsi il est impossible qu'un message ne traverse la vague de checkpoints. Un processus du système joue le rôle d'initiateur de la vague de checkpoints : il envoie un marqueur à tous les autres processus du système. Chaque processus, lorsqu'il reçoit ce marqueur, arrête son exécution, envoie un marqueur à tous les autres processus du système (on parle de *checkpoint request*) et effectue sa prise de checkpoint. Les processus ne reprennent leur exécution qu'une fois qu'ils ont reçu les marqueurs de tous les autres processus. Les canaux de communications étant FIFO, il est impossible qu'un message double le marqueur ou inversement, et l'état global sauvegardé forme une coupe cohérente. Une prise de checkpoint coordonné bloquant est illustrée figure 6 : en bleu, on voit l'exécution des processus. Le processus 1 est l'initiateur du checkpoint, c'est donc lui qui envoie le premier marqueur (en rouge).

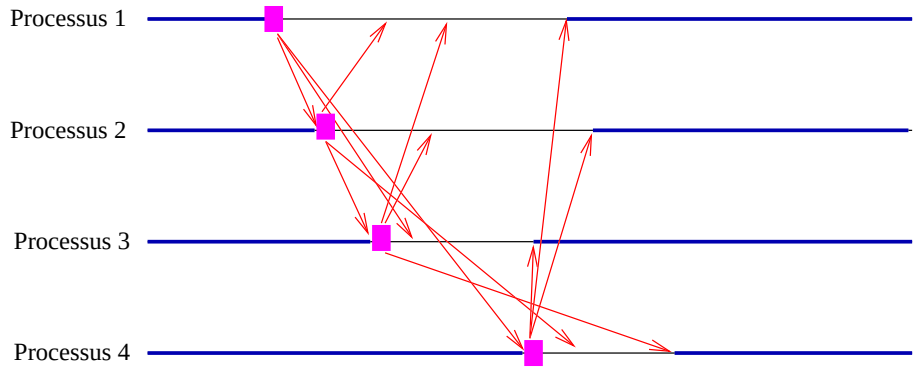


FIG. 6 – Circulation de marqueurs pour la prise d'un checkpoint bloquant

Les protocoles de checkpoint non bloquants n'arrêtent pas leur exécution au moment de la prise du checkpoint mais enregistrent les messages qui circulent entre le moment où ils prennent leur checkpoint et celui où ils ont reçu les marqueurs de tous les autres processus.

D'après [15], les protocoles de checkpoints bloquants induisent une latence plus importante et sont donc moins performants que les checkpoints non bloquants. La première partie de ce stage a concerné une implémentation du protocole de Chandy-Lamport bloquant afin de comparer ses performances avec le protocole non-bloquant (voir section 4).

Avantages et inconvénients

Après une panne, tous les processus du système redémarrent de l'état global précédent : il est impossible de rencontrer l'effet domino dans les protocoles de checkpoints coordonnés. De plus, il n'y a pas besoin de conserver en

mémoire plus de deux images globales du système, réduisant l'espace de stockage nécessaire par rapport à d'autres protocoles et simplifiant le mécanisme de nettoyage des checkpoints inutiles.

Cependant, ces protocoles introduisent une latence lors des interactions du système avec le monde extérieur. En effet, si par exemple on effectue une écriture dans un fichier, ou que l'on demande à l'utilisateur d'effectuer une saisie clavier, on ne peut pas avoir à refaire cette action en cas de retour en arrière. La solution est alors d'effectuer une prise de checkpoints après chaque entrée/sortie, ce qui introduit une latence importante dans les applications comportant beaucoup d'entrées/sorties.

En pratique

En pratique, l'implémentation de l'algorithme de Chandy-Lamport non-bloquant sur MPICH par le projet MPICH-V est MPICH-*vcl*. Cette implémentation est légèrement différente de l'algorithme de Chandy-Lamport décrit ci-dessus puisque l'initiateur de la vague de checkpoints est un élément externe au calcul appelé *checkpoint scheduler*. C'est également lui qui s'assure que la vague de checkpoints est terminée et envoie un signal à tous les noeuds de calcul afin que ceux-ci puissent exécuter le mécanisme de ramasse-miette en effaçant la vague de checkpoint précédente. Les checkpoints sont sauvegardés à distante sur un *checkpoint serveur*, qui se charge également de restituer les checkpoints au moment de la reprise de l'exécution après une faute. On suppose ces deux composants stables. Cette architecture est représentée figure 7.

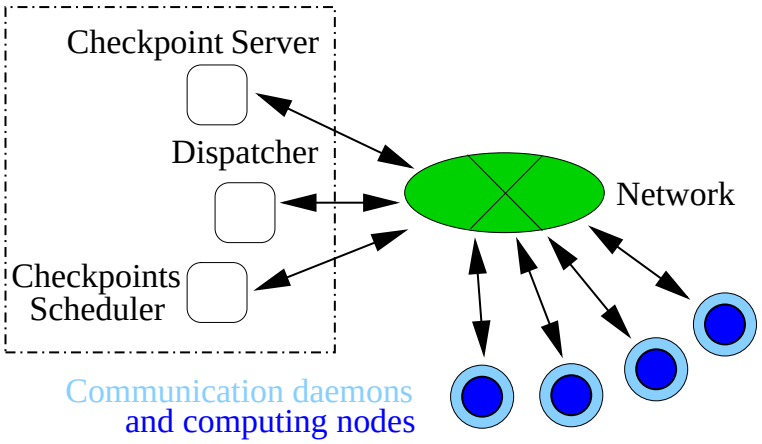


FIG. 7 – Déploiement de MPICH-*vcl*

Pour améliorer les performances lors des redémarrage, chaque processus conserve une copie en local du checkpoint qu'il vient d'effectuer afin de redémarrer à partir de la copie locale plutôt que de la redemander au serveur. Ainsi, on diminue le niveau de sollicitation du checkpoint server qui n'est alors sollicité que par le processus concerné par la panne, et le redémarrage des processus non concernés par la panne est plus rapide, l'accès à la copie locale étant plus rapide que l'accès à l'image sur le checkpoint server. Le fait d'enregistrer l'image du processus à la fois sur un support distant et en local représente un délai supplémentaire négligeable devant le temps d'enregistrement sur le support distant [29].

Une implémentation bloquante de l'algorithme de Chandy-Lamport est présentée section 4 et comparée avec l'implémentation non-bloquante de MPICH-*vcl*.

2.4.2 Message logging

La *piecewise deterministic asumption* (PWD)

Un programme est dit *déterministe* si, à partir d'un état initial donné, on arrive toujours au même état final.

Un processus faisant partie d'un système réparti communiquant avec les autres processus par passage de messages a une exécution contituée d'une succession d'évènements non-déterministes entre lesquels l'exécution est déterministe. Dans une application MPI, les évènements non-déterministes sont les réceptions de messages. Ainsi, à partir de l'état d'un processus juste après la réception d'un message, on arrive toujours au même état au moment de la réception du message suivant.

L'hypothèse selon laquelle si l'on rejoue les mêmes évènements non-déterministes, alors l'état final du système sera toujours le même pour un état initial donné s'appelle la *piecewise deterministic asumption* ou PWD [37].

Message Logging

Les protocoles de message-logging sont des protocoles de checkpoints non-coordonnés : chaque procesuss effectue ses checkpoints indépendamment des autres processus. Au cours de l'éxécution, chacun enregistre les messages échangés avec les processus afin de pouvoir effectuer à nouveau les réceptions en cas de retour en arrière.

En cas de panne, on effectue un retour en arrière et on rejoue les évènements non-déterministes (réceptions de messages). Ainsi, en vertu de la PWD, on se retrouve dans le même état qu'avant la panne.

Le fait de rejouer les messages évite l'apparition de *processus orphelins*, c'est-à-dire de processus en attente d'un message qui ne sera jamais envoyé car son expéditeur est dans son exécution au-delà du moment où il doit envoyer ce message.

Les protocoles de message-logging peuvent être basés sur l'expéditeur ou sur le destinataire (*sender-based* ou *receiver-based*) : dans le premier cas, c'est l'expéditeur d'un message qui se charge d'en assurer la sauvegarde, dans le deuxième c'est le destinataire.

Checkpoints non-coordonnés

On peut ainsi ne faire repartir de l'état initial qu'un seul processus, et faire rejouer tous les messages. De cette façon, dans un premier temps seul le processus concerné par la panne repart du début de l'exécution, et peut ensuite forcer d'autres processus à revenir en arrière eux aussi pour lui envoyer un message. En apparence, le relentissement de l'exécution lors d'une panne est alors moins important. Cependant il exite toujours des barrières de synchronisation dans les programmes par passage de message, et le processus qui a effectué un retour en arrière ralentit forcément les autres. Donc on peut sauvegarder des états intermédiaires d'où les processus repartiront en cas de panne en faisant des checkpoints, et réduire ainsi l'amplitude des retours en arrière. Contrairement aux protocoles de checkpoints coordonnés, les protocoles de checkpoints non-coordonnés n'imposent pas que tous les processus effectuent leur checkpoint au même moment. Ainsi, chaque processus prend une image de son état au moment le plus opportun.

Le principal désavantage des checkpoints non-coordonnés est le risque induit d'*effet domino*, décrit plus haut et qui peut ramener l'ensemble des processus au début du calcul. Pour éviter l'effet domino, on enregistre les messages sur un support fiable (par exemple une machine supposée stable), afin qu'ils puissent être rejoués sans nécessité de faire revenir en arrière les processus émetteurs. Trois catégories de message-logging apparaissent alors : optimiste, pessimiste et causal.

Message-logging optimiste

Les protocoles de message-logging optimistes partent de l'hypothèse qu'il

ne surviendra pas de panne entre le moment où un processus exécute un évènement non-déterministe et celui où le message est enregistré de manière fiable. Ainsi, quand un processus exécute un évènement non-déterministe, il l'envoie pour enregistrement à un support fiable et reprend son exécution sans attendre d'acquittement[37]. La latence induite est alors très faible, cependant c'est une hypothèse très forte. Si elle n'est pas réalisée, le système se retrouve après une panne dans un état inconhérent. Certains processus peuvent être orphelins, il faut alors effectuer un autre retour en arrière. On n'évite pas l'effet domino, et il faut conserver plusieurs checkpoints successifs, ce qui rend l'effacement des checkpoints qui ne seront pas utilisés plus complexe et peut rapidement nécessiter un espace mémoire important. C'est pourquoi en pratique on n'utilise pas ces protocoles.

Message-logging pessimiste

Les protocoles de message-logging pessimistes ne font pas cette hypothèse. Quand un processus exécute un évènement non-déterministe, il l'enregistre sur un support fiable et attend de recevoir un acquittement de celui-ci pour rependre son exécution [7]. En pratique, pour améliorer ces protocoles, le processus peut reprendre son exécution tant qu'il n'a pas d'impact sur le reste du système, c'est-à-dire tant qu'il n'envoie pas de message vers le reste du système. En l'absence de fautes, on a alors une latence très importante dûe au synchronisme des enregistrements des messages.

Message-logging causal

Le message-logging causal tente de combiner les avantages de ces deux familles de protocoles : peu d'overhead en l'absence de défaillances, et les processus non crashés n'ont pas à effectuer de retour en arrière (pas de création de processus orphelins, le processus crashé est isolé du reste du système). On enregistre ici encore les messages sur un support fiable, mais on n'attend pas l'acquittement pour continuer l'exécution : tant que l'acquittement n'est pas arrivé, on ajoute les messages en piggyback des messages qui sont échangés entre les processus. Ainsi, si une panne survient entre un évènement non-déterministe et son enregistrement, on peut le retrouver dans les messages qui ont été échangés avec le reste du système. Si aucun message n'a été échangé, cela n'a alors pas d'importance puisque le processus crashé n'a pas eu d'influence sur le reste du système depuis le dernier évènement sauvegardé.

Implémentation dans *Vcausal*

Si on ajoute en piggyback des messages échangés les messages précédents

dans leur intégralité, la taille des messages, piggyback inclus, devient vite très importante. De plus, si on sauvegarde chaque message échangé sur un support stable, cela équivaut à faire circuler ce message une deuxième fois sur le réseau. La charge du réseau est un élément critique pour les performances des applications par passage de messages, et il faut éviter de l'augmenter fortement comme c'est le cas ici. *Vcausal* ne sauvegarde donc pas les messages en entier sur un support externe aux processus [8]. Ils sont sauvegardés par les processus eux-même, en local. On ne sauvegarde sur un support distant que les informations de causalité entre ces messages, pour pouvoir les rejouer dans le même ordre. De même, on n'ajoute que ces informations de causalité en piggyback des messages.

Les messages sont alors identifiés de manière unique par l'émetteur comme par le receveur. Un processus qui a effectué un retour en arrière reçoit l'ensemble des déterminants des messages qu'il a reçus depuis son dernier checkpoint afin qu'il puisse demander aux expéditeurs de les rejouer dans le même ordre.

L'élément chargé de sauvegarder ces informations de causalité est appelé *event logger*.

À chaque évènement (émission ou réception), le processus incrémente une horloge locale. C'est la valeur de cette horloge qui constitue l'information de causalité, car elle donne l'ordre dans lequel des messages ont été reçus (et doivent dont être rejoués en cas de rééxécution). Pour chaque message, on sauvegarde donc un *déterminant* consituté de quatre entiers :

$$[id\_sender, H_s, H_r, nb\_prob]$$

En cas de panne, le processus concerné effectue un retour en arrière et reprend son exécution au moment du dernier checkpoint. Il obtient auprès de l'event logger et des autres noeuds les informations de causalité concernant les messages qu'il doit demander à rejouer. Il a alors connaissance de l'ordre dans lequel ils doivent être rejoués grâce aux déterminants :

- id_sender : identifiant (rang) de l'expéditeur du message
- H_s : horloge de l'expéditeur au moment de l'envoi du message
- H_r : horloge du destinataire au moment de la réception du message
- nb_prob : horloge utilisée par l'event logger

Autres protocoles de message-logging causal

D'autres protocoles de message-logging causal sont nés de l'étude des relations de causalité entre les messages envoyés dans une application par passage de messages. C'est le cas par exemple de *Manetho*[16], qui tient à jour un *graphe d'antécédence* contenant les informations de causalité entre les messages échangés.

Évaluation des protocoles de message-logging

[4] propose d'évaluer l'efficacité des protocoles à base de message-logging en sa basant sur quatre critères. Le même article propose un protocoles "optimal", dont la généralité est discutable puisque cette optimalité se base sur des critères définis dans le même article.

Nombre de retours en arrière forcés

Le nombre de processus devant effectuer un retour en arrière après une panne représente la quantité de travail perdue. La limite est 0, elle est atteinte par les protocoles pessimistes.

k-blocages

Il s'agit d'une mesure du temps perdu à ne rien faire (*i.e.,* du temps non utilisé à effectuer le calcul) qui s'ajoute à l'exécution d'un processus. k est le nombre de messages qui auraient pu être envoyés pendant ce temps, et qui sont bloqués pendant l'attente. La limite est 0, les protocoles optimistes, induisant une latence faible, étant proches de cette limite.

Nombre de messages

Il s'agit du nombre de messages additionnel envoyés par le protocole. Ce critère permet de mesurer la surcharge du réseau engendrée par le protocole. La limite est de ne pas envoyer de messages additionnels, les protocoles optimistes en étant proches.

Taille des messages

Il s'agit d'une autre mesure de la surcharge du réseau engendrée par le protocole. Si $|m_\mu|$ représente la taille des messages échangés avec le protocole de tolérance aux pannes, et $|m|$ représente la taille des messages eux-mêmes (en-dehors du protocole de tolérance aux pannes), le critère est $a = |m_\mu| - |m|$. La limite est $a = 0$, et les protocoles pessimistes basés sur le receveur sont proches de cette limite.

2.4.3 CIC

Une troisième famille de protocoles propose d'éviter l'effet domino sans nécessiter de checkpoints coordonnés : il s'agit des protocoles de *Communication Induced Checkpointing* (CIC)[26].

Ces protocoles définissent deux types de checkpoints : les *checkpoints locaux*, qui sont pris par le processus de manière indépendante, et les *checkpoints forcés*, qui sont pris pour éviter l'effet domino et garantir l'avancement de la ligne de recouvrement. En effet, un inconvénient des protocoles de checkpoints non-coordonnés est qu'ils permettent aux processus d'effectuer des checkpoints n'importe quand, y compris à des moments où il est inutile d'en faire. Comme un checkpoint ralentit l'exécution, il faut éviter d'en faire inutilement.

Contrairement aux protocoles de checkpoints coordonnés, les protocoles de CIC ne nécessitent pas d'échanges de messages spécifiques pour forcer les checkpoints : à la manière des protocoles de message-logging causal, on ajoute la signalisation en piggyback des messages échangés.

Il existe deux familles de protocoles CIC : coordonnés en se basant sur un modèle (model-based coordination) ou sur les indices (index-based coordination). Fondamentalement, il a été montré qu'ils sont équivalents. Cependant en pratique il a été constaté que la coordination basée sur les indices force moins de checkpoints.

Model-based coordination

Ces protocoles se basent sur la notion de *Z-paths* et de *Z-cycles*. Un Z-path est le chemin constitué par un ensemble de messages dont l'émission ou la réception dépendent les uns des autres. Un Z-cycle est un Z-path qui boucle. On peut voir sur la figure 8 trois processus qui communiquent par passage de messages. Les messages 1 et 2 forment un Z-path, car la réception du message 2 est conditionnée par celle du message 1. De même, l'émission du message 3 est conditionnée par la réception du message 2 : les messages 2 et 3 forment ainsi un autre Z-path. Enfin, les messages 1 et 3 forment un Z-cycle. On évite les checkpoints inutiles en évitant que les Z-paths ne se transforment en Z-cycles. On force donc un checkpoint avant que ça n'arrive : on l'effectue avant de délivrer à l'application le message qui va transformer le Z-path en Z-cycle. Un processus n'a ici pas d'information sur l'état global du système : c'est pourquoi il peut forcer plus de checkpoints que le minimum nécessaire.

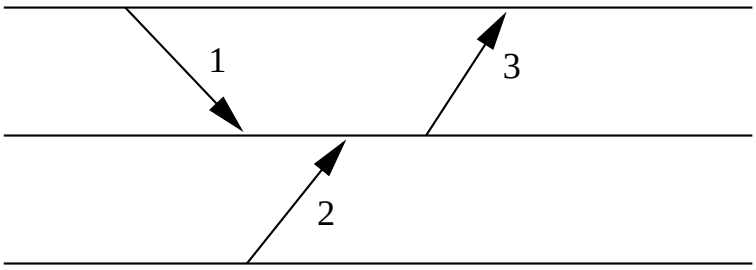


FIG. 8 – Z-paths et Z-cycles

Index-based coordination

Ces protocoles utilisent une horloge logique ajoutée en piggyback des messages, afin que leurs destinataires sachent s'ils doivent effectuer un checkpoint. Si l'on note $\prec$ la relation de précédence causale définie par Lamport [28], les checkpoints sont forcés afin de conserver les propriétés suivantes :

- Si on a deux checkponts $c_{i,m}$ et $c_{j,n}$ (i et j étant des processus, m et n des valeurs de leurs horloges logiques) tels que $c_{i,m} \prec c_{j,n}$, alors $\text{ts}(c_{i,m}) \leq \text{ts}(c_{j,n})$ (ts = timestamp)
- Les checkpoints locaux consécutifs ont une horloge logique croissante

Ces propriétés garantissent que l'ensemble des checkpoints portant la même horloge forment un état global cohérent. Par analogie avec les protocoles de CIC dont la coordination est basée sur les modèles, on peut faire croître cette horloge le long d'un Z-path : ainsi il est impossible de former un Z-cycle d'après ces propriétés.

Évaluation des protocoles de CIC

Un checkpoint prend du temps sur l'exécution d'un processus. Il vaut donc mieux en avoir à exécuter le moins possible. Or, avec les protocoles de CIC, on ne maîtrise pas la fréquence des checkpoints puisqu'ils sont effectués selon une certaine heuristique. Une évaluation de ce type de protocoles a montré qu'ils forçaient beaucoup trop de checkpoints du fait de cette heuristique, ce qui les rend moins intéressants que d'autres protocoles sur les benchmarks utilisés, qui simulent une utilisation typique d'un cluster [3].

3 MPI et ses implémentations

3.1 Message-Passing Interface

3.1.1 MPI-1

MPI-1 est une norme créée en 1994 pour le calcul à haute performances dans le but de créer une interface unique. En effet, jusque là les librairies de passage de messages étaient des logiciels propriétaires développés par les constructeurs de machines. Un consorsium formé de chercheurs et de fabriquants de calculateurs s'est alors formé afin de définir une norme unifiant ces bibliothèques, puisqu'il a alors été démontré que le fait d'utiliser une bibliothèque non pas spécifique à la machine mais plus universelle n'affecte pas de manière significative les performances de la machine. Cette norme est disponible sur une large gamme de machines parallèles, des clusters de PC aux clusters de processeurs vectoriels, ce qui rend un code MPI utilisable sur différentes architectures. La durée de vie des applications MPI existantes ne dépend plus ici des évolutions matérielles et logicielles, et est ainsi alongée. La norme MPI est vite devenue le standard *de facto* pour les applications parallèles par passage de messages.

La première version de la norme [20] définit une API C et FORTRAN 77, des communications efficaces point-à-points et collectives, les notions de groupe de processus, de contextes de communication et de topologie, et une interface pouvant être implémentée sur un grand nombre de plateformes. Elle suppose que les communications sous-jacentes sont sûres. La version 1 de la norme ne permet pas de gérer de façon sûre les processus multi-threads, tandis que la version 1.1 a introduit cette possibilité.

3.1.2 MPI-2

La norme MPI-2 [22] est une extension de MPI-1. Tout en restant compatible avec MPI-1, c'est-à-dire que les programmes développés en accord avec MPI-1 sont exécutables sous une implémentation de MPI-2, elle lui ajoute des fonctionnalités parmi lesquelles la possibilité de créer dynamiquement des processus, d'effectuer des communications unilatérales, de rattacher un processus à l'exécution ou une extention des communications de groupe.

3.2 MPICH

3.2.1 MPICH1

MPICH [24] est une implémentation de la norme MPI-1. D'une grande portabilité, elle est disponible pour Linux, Unix et Windows. Plusieurs types de réseaux sont supportés, comme les réseaux TCP, Infiniband ou Myrinet. De plus, il existe un driver permettant d'utiliser le middleware Globus.

Son architecture est composée de trois couches principales :
- La couche d'abstraction qui présente les routines MPI de haut niveau.
- Un API qui présent les routines de communications appelée *chameleon*.
- Une couche de bas niveau (driver) dépendant du type de réseaux utilisé.

La particularité de MPICH sur Ethernet vient du driver pour réseaux Ethernet, p_4, où les communications sont gérées par un démon qui délivre les messages à l'applications. Ce démon n'existe plus dans MPICH2.

3.2.2 MPICH2

MPICH2 est une implémentation de la norme MPI-2. Son architecture est basée sur les mêmes couches, l'API *chameleon* ayant été redéfinie. Les réseaux Ethernet sont gérés par un driver différent, SOCK, n'impliquant pas de démon à la manière de p_4. L'environement d'exécution est lui aussi différent.

3.3 LAM/MPI

LAM/MPI [11], projet aujourd'hui abandonné, implémente un mécanisme de tolérance aux fautes utilisant un algorithme de Chandy-Lamport bloquant [32] : à la réception d'un signal, chaque processus vide ses canaux de communications avant de prendre un checkpoint. Cette synchronisation entre les processus permet d'obtenir un snapshot global représentatif d'un état global cohérent. Elle est faite en utilisant des fonctions MPI, comme la barrière de synchronisation qui synchronise tous les processus au moment d'effectuer une checkpoint.

3.4 Open MPI

Open MPI [21] est le projet le plus récent cité ici. Il réunit les équipes de LAM/MPI, LA-MPI et FT-MPI. Son but est de fournir une implémentation complète de la norme MPI-2 offrant les meilleures performances possibles. L'architecture d'Open MPI est composée de trois composants :

- La couche Open MPI (OMPI), qui constitue l'API MPI de haut niveau
- L'environement d'exécution (ORTE, *Open Run-Time Environment*), qui est une interface vers le driver
- Le driver (OPAL, *Open Portability Access Layer*)

Il s'agit de *dépendances* et non de couches d'abstraction. On a les relations de dépendances suivantes : $OMPI \rightarrow ORTE \rightarrow OPAL$. L'architecture d'Open MPI est schématisée figure 9.

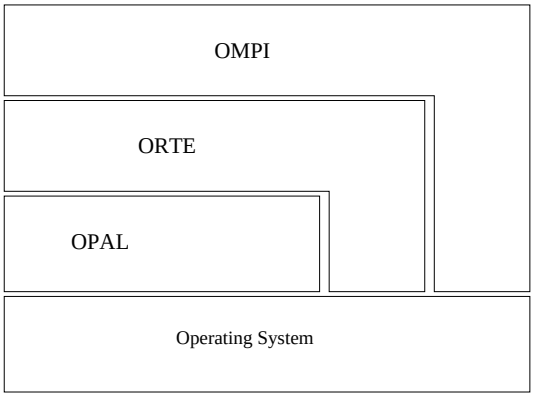


FIG. 9 – Architecture d'Open MPI

Les défaillances sont gérées par un sous-système de ORTE. Pour le moment, en cas de défaillance, le comportement adopté consiste à arrêter l'exéution de tous les processus. Cependant il est prévu d'intégrer un mécanisme de gestion des erreurs dans le futur.

3.5 Autres implémentations

Il existe d'autres implémentations de MPI, dont certaines sont tolérantes aux fautes.

CoCheck [36] est utilisé au-dessus d'un système par passage de messages, et peut ainsi être facilement adapté d'un système à un autre.

Starfish [2] est un enrichissement de l'API MPI. De nouvelles instructions sont rajoutées au langage afin de permettre à l'utilisateur de choisir la stratégie de tolérance aux fautes qu'il désire utiliser.

Clip [14] propose lui aussi de nouvelles instructions. Il s'agit d'une tolérance aux fautes semi-transparente : c'est le programmeur qui spécifie dans le code de son application quand les processus doivent effectuer les checkpoints.

FT-MPI [18] est une implémentation de la norme MPI-1.2 reposant sur un environement d'exécution tolérant aux pannes [19].

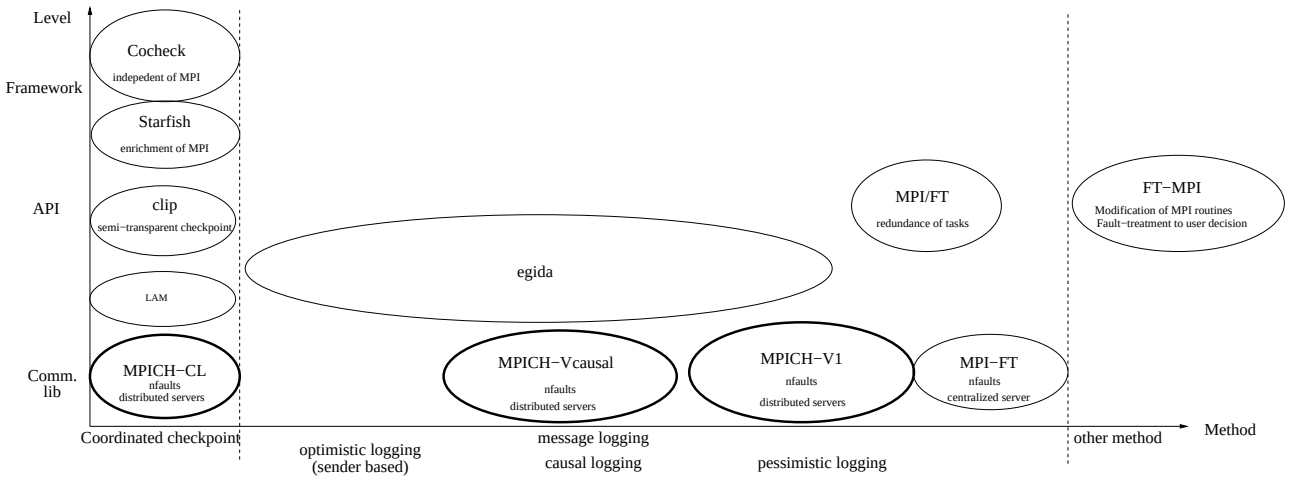


FIG. 10 – Tableau récapitulatif des bibliothèques MPI tolérantes aux fautes, techniques de tolérance aux fautes et niveaux d'implémentation.

4 MPICH2-*pcl*

4.1 Algorithme

Le protocole *pcl* (pour *Preemptive Chandy-Lamport*) utilise l'algorithme de Chandy-Lamport. La synchronisation des processus au moment de la prise du checkpoint est faite en vidant les canaux de communication. On s'assure qu'il n'y a pas de message transitant entre les processus au moment de la prise de checkpoint. Pour ce faire, chaque processus envoie un marqueur aux autres processus avant d'effectuer son checkpoint. Quand un processus reçoit un tel marqueur, il envoie lui aussi un marqueur aux autres processus et effectue un checkpoint. Ainsi, en faisant l'hypothèse que les canaux de communication ont la propriété FIFO, on s'assure que l'état représenté par le snapshot global du système constitue un état cohérent. Ce protocole et les protocoles de retour arrière sur points de reprise coordonnés sont présentés section 2.4.1.

4.2 Architecture et implémentation

4.2.1 Prise des checkpoints

Il existe deux niveaux d'abstraction permettant la prise de checkpoint d'un processus :

- Niveau système : on sauvegarde l'état de la pile, des registres et le contenu de la mémoire virtuelle [38] [27] [30]
- Niveau applicatif : on enregistre l'endroit où on est dans le programme et des valeurs des variables [10]

On effectue ici la prise de checkpoints au niveau système. Pour cela, on utilise une bibliothèque appelée Berkeley Linux Checkpoint/Restart (BLCR). L'avantage de cette approche est qu'elle ne nécessite pas de changement dans le code de l'application.

La prise de checkpoints au niveau système présente l'inconvénient de produire des volumes importants de données, ce qui représente un trafic réseau très important au moment des transferts de checkpoints. La prise de checkpoints au niveau applicatif n'est pas transparente du point de vue du programmeur, qui doit spécifier dans son code les endroits où sont pris les checkpoints. C'est pourquoi la prise de checkpoints au niveau applicatif ne remplit pas les conditions de transparence.

4.2.2 Architecture du système

Chaque noeud MPI est constitué de deux processus : le noeud de calcul lui-même, et le *process manager*. Le process manager a pour rôle de fournir les informations sur la localisation des noeuds pour envoyer les messages.

L'environnement d'exécution de l'application est le *mpiexec*. Il prend en argument le fichier contenant les noms des machines à utiliser, le nombre de processus MPI à lancer et le nom de l'exécutable (à la manière de *mpirun* pour MPICH). C'est lui qui lance les noeuds MPI et les serveurs de checkpoint à partir du fichier de machines fourni en argument.

L'architecture de MPICH2-*pcl* est représentée figure 11.

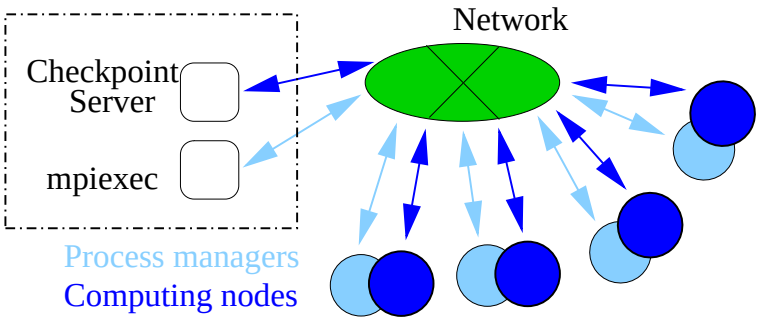


FIG. 11 – Déploiement de MPICH2-*pcl*

4.2.3 Implémentation dans MPICH2

L'architecture de MPICH2 est constituée de plusieurs couches. Un schéma représente cette architecture figure 12.

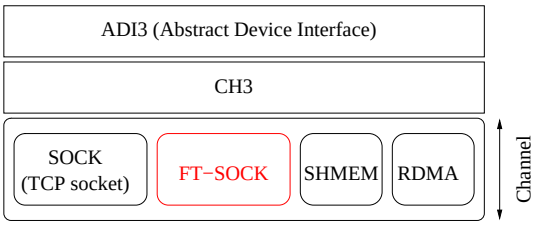


FIG. 12 – Architecture de MPICH2

Le niveau d'abstraction le plus bas est le *canal*. Il s'agit d'un ensemble de drivers pour s'adapter aux différents réseaux. Il existe un driver pour les réseaux TCP, pour Myrinet... Ici la tolérance aux pannes est implémentée dans le driver *FT-SOCK*, qui est une extension du driver pour réseaux TCP *SOCK*.

Au-dessus du canal se situe *chameleon 3* (CH3) qui permet l'abstraction de la couche supérieure en fournissant les routines de communication sous la forme d'une API.

La couche la plus haute est l'*Abstract Device Interface* (ADI). C'est elle qui lie les instructions MPI aux routines de communication de haut niveau.

4.3 Le serveur de checkpoints

Le serveur de checkpoint reçoit les fichiers contenant les snapshots résultant de la prise de checkpoint de chaque processus. C'est également lui qui, après une défaillance, envoie le fichier demandé nécessaire à la restauration d'un processus. On suppose qu'il est situé sur une machine stable par rapport aux autres machines du système. Le développement du serveur de checkpoint et de la partie client correspondante a constitué la première partie de mon stage. Il s'agit d'un serveur générique pouvant être utilisé avec le protocole *pcl* ou *vcl*, c'est pourquoi j'ai aussi implémenté les spécificités pour *vcl* comme la sauvegarde des messages.

Architecture client-serveur

Quand un processus s'apprête à effectuer un checkpoint, il se duplique. Le processus fils se duplique également. C'est le petit-fils qui effectue la prise du checkpoint. Comme chaque processus fils est dupliqué à l'identique de son processus père, le fait qu'il s'agisse du petit-fils n'a pas d'influence sur le checkpoint.

Le petit-fils écrit son snapshot dans un fichier, puis le fils envoie au serveur de checkpoints. Le fils est donc le client du serveur de checkpoints.

Une fois le checkpoint pris et écrit entièrement dans le fichier, le petit-fils termine son exécution. Le fils reçoit alors le signal *SIGCHLD*, et sait que le fichier contient le snapshot dans son intégralité.

Dans un premier temps, il était prévu de pipeliner la prise du checkpoint et son envoi, c'est-à-dire que le client envoyait le snapshot au serveur de checkpoint au fur et à mesure que le processus petit-fils prenait le checkpoint et l'écrivait dans le fichier. C'est possible dans l'hypothèse où l'accès au disque dur local est plus lent que l'accès au réseau. Cependant, lorsque le nombre de noeuds effectuant l'envoi de leur snapshot simultanément devient important, cette hypothèse n'est plus vraie et le temps d'accès au réseau devient supérieur au temps d'accès au disque local. C'est pourquoi cette optimisation a été abandonnée, et le snapshot n'est envoyé que lorsque le checkpoint a été entièrement effectué (donc lorsque le petit-fils meurt).

Le mécanisme de prise et d'envoi du checkpoint est représenté figure 13.

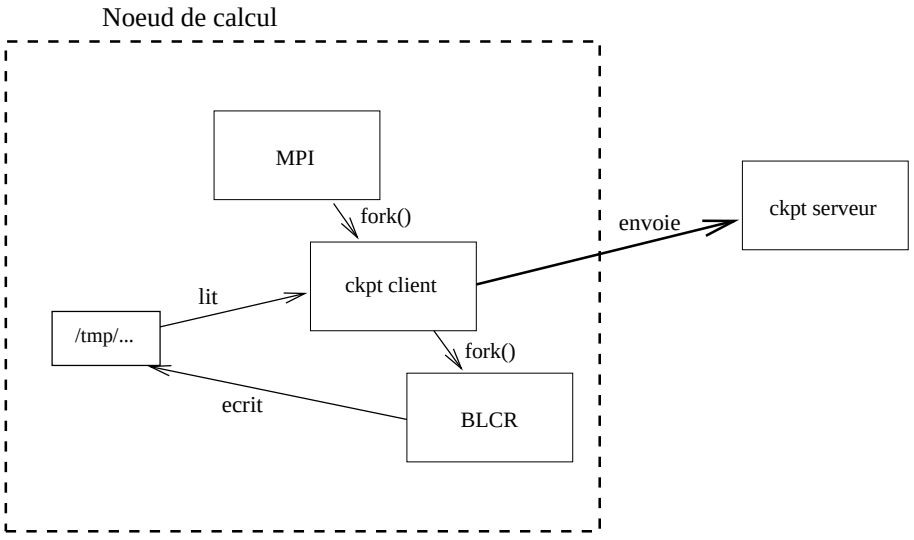


FIG. 13 – Fonctionnement de la partie client : prise et envoi d'un checkpoint

Communications client-serveur

La signalisation des communications se fait au moyen d'un paquet envoyé par le client contenant les informations nécessaires. On transmet une structure de type `signalisation` contenant 3 champs :

- `type` : 's' (resp. 'g') pour "set" (resp. "get") pour signifier l'envoi (resp. la requête d'envoi) d'un checkpoint
- `rank` : le rang du client
- `phase` : identifiant du checkpoint, *i.e.,* indice de la vague de checkpoint

Le serveur reçoit ce paquet et détermine, en fonction du champ `type`, s'il doit se préparer à recevoir un checkpoint ou s'il doit en envoyer un.

Fonctions du serveur

Le serveur implémente deux fonctions : `set_snapshot()` (recevoir un checkpoint d'un client) et `get_snapshot()` (envoyer un checkpoint à un client).

Prise d'un checkpoint

Lors de l'envoi d'un checkpoint, il est possible que le client ne connaisse pas la taille totale du snapshot au moment du début de l'envoi suivant la manière dont il a été implémenté. C'est le cas si le client lit le snapshot dans un fichier pendant qu'il l'envoie au fur et à mesure et donc ne connaît pas la taille avant d'avoir lu la fin du fichier. Cependant, il est impossible de transmettre le snapshot et la taille sur la même socket. Le client a donc besoin de deux sockets distinctes. La première est la socket de contrôle, sur laquelle il doit envoyer les informations de contrôle. Il s'agit de la socket connectée au serveur. Le serveur créé la seconde, destinée à recevoir le snapshot, lorsqu'il s'apprête à recevoir le snapshot. Si ce serveur est utilisé pour le protocole *vcl*, il créé une troisième socket pour recevoir les messages.

Ces sockets sont alors multiplexées dans un appel système *poll* qui écoute ce qui vient des deux (ou trois) sockets à la fois.

Le serveur reçoit le snapshot sur sa socket de données. Il ne connaît pas forcément sa taille totale à l'avance. C'est pourquoi un algorithme d'adaptation de la taille du buffer est implémenté ici. On commence par lire un nombre donné d'octets (fixé en dur dans le programme) et on les met dans un tampon de la taille correspondante. Si ce nombre est insuffisant, on le double, on réalloue de la mémoire dans le tampon et on recommence à lire ce qui vient sur la socket.

À chaque fois que l'on lit des données sur la socket, on ajoute le nombre d'octets lus à un compteur. Une fois que le nombre total d'octets lus sur la socket de données est égal à la taille totale du snapshot (reçue sur la socket de contrôle), on écrit le snapshot dans un fichier et on envoie un acquittement au client sur la socket de contrôle.

Nettoyage des anciens checkpoints

En cas de panne, le retour se fait sur la dernière vague de checkpoints effectuée correctement. Il n'est donc pas nécessaire de conserver de checkpoints antérieurs à cette dernière vague. Une défaillance peut survenir lorsqu'une vague de checkpoints est en train d'être effectuée, auquel cas le snapshot

global du système ne sera pas complet. Il faut dont conserver la vague précédente jusqu'à ce que la vague de checkpoints soit entièrement effectuée. Or, si l'on utilise plusieurs serveurs de checkpoints, chaque serveur ne sait pas si les autres ont correctement reçu les snapshots de tous leurs clients. Le seul moment où un serveur de checkpoints sait que la vague précédente a été effectuée correctement dans son intégralité est lorsqu'il commence à recevoir les snaphots d'une nouvelle vague. En effet, une nouvelle vague n'est pas commencée avant que la dernière soit terminée. Les serveurs de checkpoints conservent donc en permanence entre une et deux vagues successives : lorsqu'un snapshot correspondant à la vague n arrive, le serveur efface les fichiers contenant les snapshots de la vague $n-2$. Ainsi, en cas de défaillance pendant la prise de la vague n, un retour en arrière peut être effectué sur la vague $n-1$.

Envoi d'un checkpoint

Lorsqu'une panne survient, les processus qui ne sont pas morts pendant la panne effectuent un retour en arrière sur la copie du snapshot qu'ils ont conservé en local. Comme expliqué section 2.4.1, il s'agit d'une optimisation destinée à diminuer le stress du serveur de checkpoints au moment du retour en arrière. Le processus qui a subi directement la panne récupère l'image de son état lors de la dernière vague de checkpoints terminée avant le panne auprès du serveur de checkpoints.

Le client envoie sa demande en précisant son rang dans le message de signalisation envoyé au serveur. Le serveur lui envoie le snapshot correspondant à son rang et faisant partie de la dernière vague de checkpoints reçue complètement, c'est-à-dire pour laquelle il a reçu entièrement les checkpoints de tous les autres processus.

De même que pour la réception d'un snapshot, le serveur a besoin de 2 sockets pour *pcl*, 3 pour *vcl* : une socket de contrôle, une socket de données et éventuellement une socket pour les messages.

Lorsqu'il lit le snapshot dans le fichier, le serveur ne connaît pas sa taille avant d'avoir lu le fichier intégralement. C'est pourquoi il a besoin ici aussi d'une socket de contrôle où il transmet la taille totale du snapshot une fois qu'il l'a calculée. Le serveur lit un nombre donné d'octets dans le fichier (fixé en dur dans le code), les copie dans un tampon et transmet son contenu au client. Une fois qu'il a tout envoyé, le serveur attend l'acquittement du client.

4.4 Comparaison *pcl* vs *vcl*

4.4.1 MPICH-*vcl*

Présenté dans la section 2.4.1, MPICH-*vcl* est une implémentation non-bloquante de l'algorithme de Chandy-Lamport : les messages qui transitent entre les processus lors d'une vague de checkpoints sont sauvegardés. Ainsi, lors d'un retour en arrière, les messages traversant une vague de checkpoints (*i.e.,* dont l'envoi et la réception sont situés de part et d'autre de la vague de checkpoints) peuvent être rejoués et l'état représenté par le snapshot global et l'ensemble des messages est cohérent.

4.4.2 Expérimentations

Conditions expérimentales

Nous avons testé MPICH-*vcl* et MPICH2-*pcl* dans trois conditions :

- Cluster à hautes performances : des machines dans un même cluster, reliées par un réseau à hautes performances (Myri2000.
- Grappe de PC : des machines dans un même cluster, reliées par un réseau GigaEthernet.
- Grille : plusieurs grappes de PC reliées entre elles.

Il est à noter que les expériences sur réseau Myrinet ont été réalisées en utilisant une émulation de réseau TCP sur Myrinet. En effet, nous n'avons pas implémenté les protocoles de tolérance aux pannes sur d'autres drivers que celui destiné aux réseaux TCP. Les performances sont donc légèrement diminuées par cette émulation.

Nous avons utilisé la plate-forme expérimentale Grid5000, en particulier les clusters de Bordeaux, Lille, Orsay, Rennes, Sophia-Antipolis et Toulouse. Ils sont tous constitués de machines bi-processeurs équipées d'AMD Opterons cadencés à 2 GHz, avec chacun 20 Go de swap, reliés par un réseau Gigabit Ethernet. Si Grid5000 propose d'autres architectures, nous nous sommes limités à une seule par soucis d'homogénéité. Les clusters sont reliés par de la fibre noire.

Grid5000 permet d'utiliser son propre environnement, en créant un système de fichiers et en y copiant une image de système d'exploitation au moment du démarrage des noeuds. Ainsi, chaque utilisateur peut utiliser le système d'exploitation de son choix, sous Linux il peut choisir sa distribution, etc. Nous avons utilisé un système GNU/Linux Debian sur un noyau 2.6.13.5 et le compilateur GCC-4.0.3. Nous avons utilisé dans cet environnement la

bibliothèque Berkeley Linux Checkpoint/Restart Library (BLCR [27]), qui permet d'effectuer des checkpoints au niveau système.

Prise des checkpoints

La prise des checkpoints est commandée par un chronomètre. Afin de mettre en évidence l'impact de la prise et de l'envoi des checkpoints sur l'exécution des applications, ils sont effectués à une fréquence particulièrement élevée (espacés de quelques dizaines de secondes).

Benchmarks

Nous avons utilisé des benchmarks issus de la suite NAS (NPB-2.3, NAS Parallel Benchmark), de la NASA [5], qui représente une application parallèle suivant un modèle de communications classique. En particulier, nous en avons utilisé deux (BT et CG) présentant des communications différentes. De plus, nous avons utilisé deux tailles (B et C) afin de jouer sur la complexité des calculs à effectuer.

Les benchmarks BT de classe B et C communiquent peu et par de longs messages, mettant ainsi en évidence les performances de calculs plutôt que les communications.

Le benchmark CG effectue des communications de petite taille afin de mettre en évidence la latence du système.

Communications entre les clusters

Nous avons testé la bande passante et la latence des communications entre les clusters grâce à l'utilitaire NetPIPE [35]. On remarque figure 14 le fait que les communications entre deux noeuds d'un même cluster sont bien plus performantes tant en matière de latence que de bande passante qu'entre deux noeuds de deux clusters différents. Le réseau est 20 fois plus rapide à l'intérieur d'un cluster qu'entre deux clusters, et la latence est plus importante de deux ordres de grandeur.

Résultats à grande échelle

Les expériences au-delà de 300 noeuds n'ont pu être menées qu'avec MPICH2-*pcl*, MPICH-*vcl* ne fonctionnant pas à cette échelle. En effet, les sockets sont multiplexées en utilisant l'appel système `select`, qui sous Linux ne supporte pas plus de 1024 sockets. Chaque noeud étant connecté au process manager au moyen de 3 sockets, le nombre de processus est alors limité.

	Bordeaux		Orsay		Rennes		Sophia		Lille		Toulouse	
	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)	BW (Mb/s)	Lat (ms)
Toulouse	190	1.5	59.81	5.51	34.79	9.92	83.7	3.74	26.97	13.04	930.4	0.04
Lille	30.23	11.62	132.55	2.25	56.84	5.83	37.41	9.22	938	0.04		
Sophia	68.1	5.2	42.4	8.6	40.2	9.1	940.5	0.04				
Rennes	110.1	4.0	95.4	4.7	940.4	0.04						
Orsay	108.1	4.1	930.4	0.06								
Bordeaux	940.2	0.04										

FIG. 14 – Évaluation des performances des communications sur Grid5000

On peut voir figure 15 l'évolution du temps d'exécution du benchmark BT de classe B en fonction du nombre de processus impliqués. Nous avons utilisé ici 4 serveurs de checkpoints par cluster, chaque noeud de calcul utilisant un serveur faisant partie du même cluster que lui. On constate une augmentation importante du temps d'exécution à 529 processeurs. Cela s'explique premièrement par le fait que le benchmark n'ait pas été conçu pour de telles échelles : il s'agit d'un benchmark de cluster et non pas de grille. Cependant nous l'avons utilisé car il constitue tout de même un bon test pour notre protocole en le soumettant à des schémas de communications complexes. La deuxième raison est l'hétérogénéité de la grille, et la nécessité d'utiliser des processeurs distants qui ralentissent l'exécution. Le nombre de vagues de checkpoints effectuées étant proportionnel au temps d'exécution (une vague toutes les 60 secondes), on effectue alors 6 vagues de checkpoints, contre 5 pour les autres mesures. Chaque vague de checkpoints ralentissant l'exécution, le temps d'exécution s'en trouve d'autant plus augmenté.

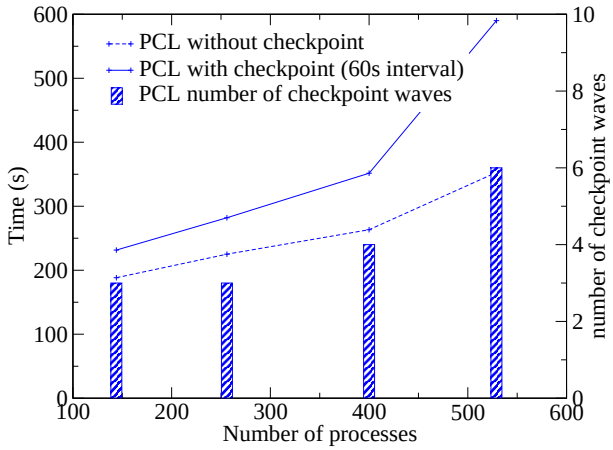


FIG. 15 – Passage à l'échelle de MPICH2-*pcl*

La figure 16 représente l'impact de la fréquence de prise des checkpoints sur le temps d'exécution de l'application et confirme ce ralentissement. Nous avons ici fait varier la fréquence des vagues de checkpoints pour un nombre de processeurs fixé à 400 exécutant le benchmark BT de classe B. On constate que la durée de l'exécution varie linéairement avec le nombre de vagues de checkpoints, même pour une exécution sur grille (et non plus sur un seul cluster).

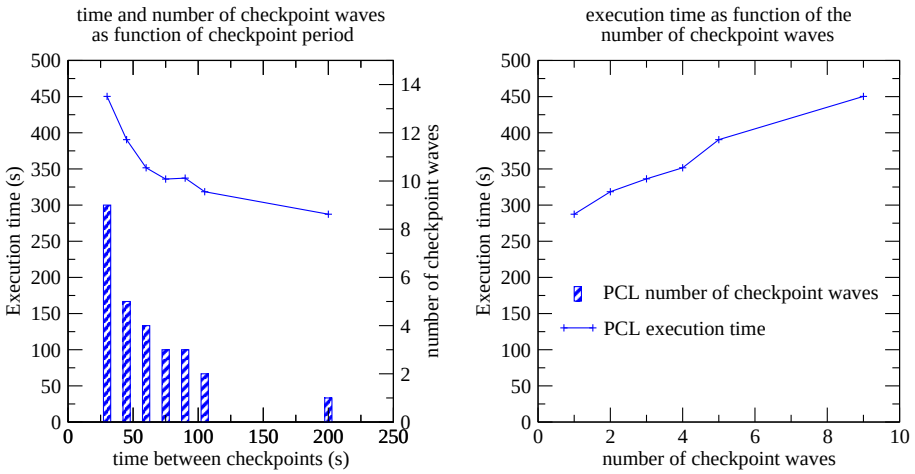


FIG. 16 – Impact de la fréquence des checkpoints

MPICH2-*pcl* est stable à grande échelle : nous avons réussi à mener une expérience sur 1 024 noeuds. Cependant, du fait du manque de disponibilité de Grid5000, nous n'avons pas pu obtenir de mesure fiable. Cette expérience nous a permis de nous rendre compte que, pour mener des expériences à cette échelle, le mpiexec a besoin d'être reconçu. En effet, les noeuds sont connectés en *one-to-one*, soit 523 776 connexions. Le processus mpiexec crée un thread par connexion, et utilise alors, pour 1024 noeuds, environ 3 Go de mémoire, ce qui est beaucoup trop. De plus, les quantités de données transférées sur le réseau lors des enregistrements des checkpoints sont alors très importantes, et une vague de checkpoints prend trop de temps. Les recherches sur le checkpointing au niveau applicatif visent à réduire la taille d'un checkpoint et donc de réduire le volume de données à transférer, cependant à l'heure actuelle ces techniques ne remplissent pas la contrainte de transparence [34] [9].

Le protocole *Pcl* nécessite une synchronisation sur l'ensemble des procesus au moment de la vague de checkpoints qui, sur une grille, induit un ralentissement important du fait de la lenteur des communications entre les clusters

(figure 14). C'est pour cette raison que ce protocole n'est pas adapté à une environnement de grille à grande échelle.

Résultats sur grappe de PC

Nous avons ensuite comparé *Pcl* et *Vcl* sur des grappes de PC. Il s'agit des machines Opteron de Grid5000, reliées par un réseau Eternet Gigabit. Nous avons ici utilisé le benchmark BT de classe B sur 64 processus répartis sur 32 machines bi-processeurs, en déclenchant une vague de checkpoints toutes les 30 secondes. Nous nous sommes intéressés ici à la scalabilité de serveur de checkpoint, c'est-à-dire au fait que l'augmentation du nombre de serveurs de checkpoints diminue de manière significative ou non la durée totale de l'exécution.

La partie haute de la figure 17 montre le temps d'exécution de l'application en fonction du nombre de serveurs de checkpoints utilisé. On constate que ce temps diminue quand on augmente le nombre de serveurs avec MPICH2-*Pcl*, tandis qu'il reste constant avec MPICH-*Vcl*. La partie basse de la figure montre le nombre de vagues de checkpoints effectuées durant l'exécution. Ce nombre reste constant avec MPICH-*Vcl*, tandis qu'il diminue comme le nombre de serveurs augmente avec MPICH2-*Pcl*. Ceci s'explique par le fait que *Pcl* étant un protocole bloquant, le transfert des checkpoints et les communications de l'application ne se gênent pas : le transfert des checkpoints ne ralentit pas les communications de l'application, et inversement. Si le nombre de serveurs de checkpoints augmente, la sauvegarde des checkpoints se fait plus rapidement, et le temps de checkpoint est réduit. L'exécution peut reprendre plus rapidement, et n'est alors pas ralentie. Au contraire, *Vcl* étant un protocole non bloquant, les transferts de checkpoints sont en concurrence avec les communications de l'application, ce qui ralentit l'exécution de l'application et la prise de checkpoints.

On peut voir figure 18 la scalabilité sur un cluster de MPICH-*Vcl* et MPICH2-*Pcl* suivant la fréquence des checkpoints. Le nombre de serveurs de checkpoints est fixé à 9. Sans surprise, les temps d'exécution sans prise de checkpoints sont plus courts qu'avec des prises de checkpoints durant l'exécution.

Le ralentissement observé à partir de 169 processeurs provient du fait que nous avons utilisé 144 machines bi-processeurs, en affectant à partir de cette taille deux processus par machine. Si les performances en calculs ne sont pas affectées, les processus devaient alors partager la carte réseau de leur machine, ce qui pénalisait alors les performances des communications.

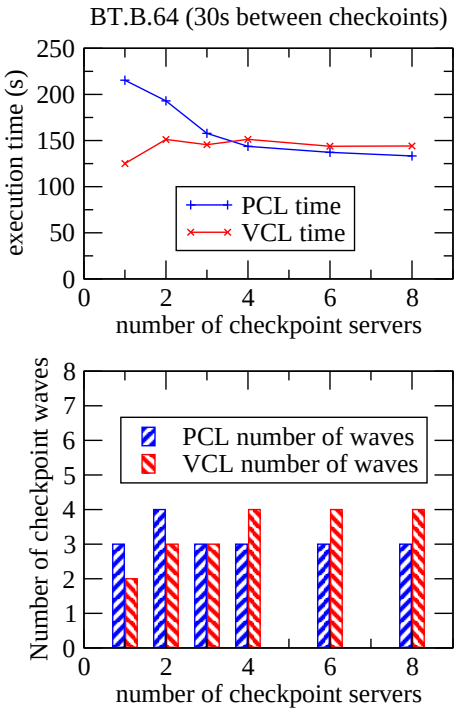


FIG. 17 – Temps d'exécution d'une application en fonction du nombre de serveurs de checkpoints

On voit que pour des fréquences de checkpoints importantes, les performances de ce protocole se dégradent, les prises des checkpoints perturbant les communications et les phases de synchronisation prenant trop de temps sur l'exécution. En-dehors du cas extrême où les checkpoints sont pris toutes les 10 secondes, on constate que le nombre de processus impliqués n'a pas d'influence sur la durée de l'exécution. On constate également que l'augmentation du temps entre les checkpoints diminue la différence de performances entre les deux protocoles.

Résultats sur cluster à communications à hautes performances

Nous avons enfin observé les performances des deux protocoles sur un réseau myri2000. Nous avons pour cela utilisé le benchmark CG de classe C sur 64 processeurs et le benchmark BT de classe B sur 64 processus. Les deux séries de mesures ont été menées sur une grappe de 36 machines interconnectées par un réseau myri2000. La tolérance aux pannes n'étant pas implémentée pour le driver pour réseaux Myrinet, nous avons utilisé le driver MX-2G 1.1.1 émulant un réseau Ethernet sur un réseau myri2000. Pour toutes les mesures nous avons utilisé 4 serveurs de checkpoints.

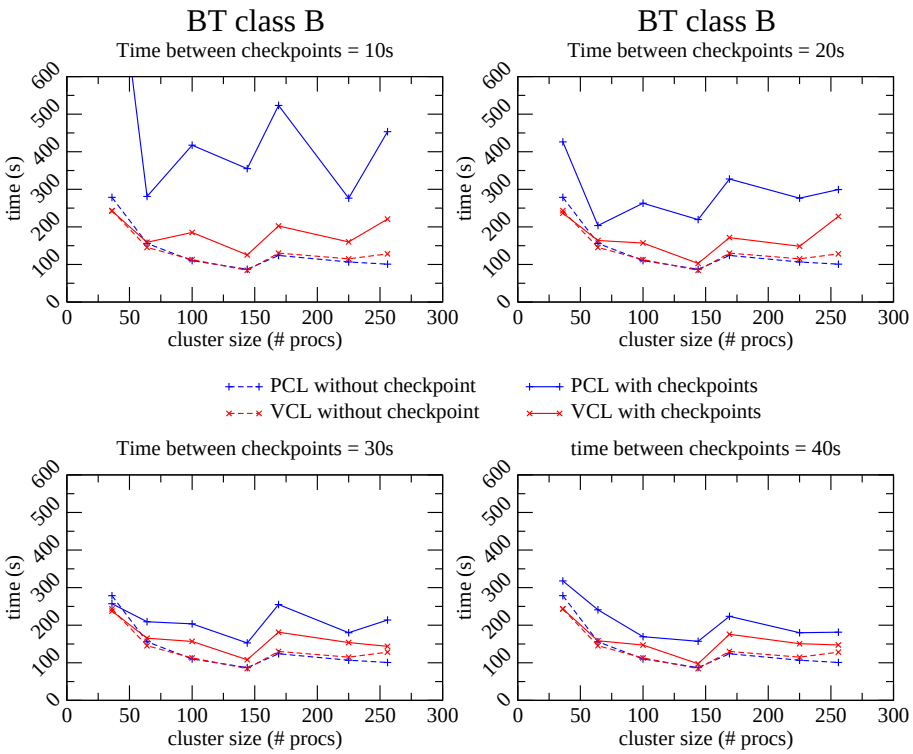


FIG. 18 – Scalabilité pour différentes fréquences de checkpoints

On peut voir figure 19 sur les courbes de gauche le temps d'exécution des benchmarks en fonction du temps entre deux checkpoints, et à droite le temps d'exécution en fonction du nombre de vagues de checkpoints. On constate sur ces dernières courbes que le temps d'exécution pour *Pcl* est sensiblement linéaire par rapport au nombre de checkpoints effectués. Cette linéarité s'explique par le mécanisme de synchronisation du protocole, qui fait qu'une vague de checkpoints prend toujours le même temps. Comme constaté avec les mesures sur grappe de PC, le nombre de vagues de checkpoints avec le protocole *Vcl* n'a pas d'influence directe sur la durée de l'exécution en raison du fait que les transferts des checkpoints se font en concurrence avec les communications de l'application.

Le protocole *Vcl* est implémenté avec un démon de communications. Chaque message doit alors passer par deux sockets UNIX, ce qui implique des copies supplémentaires de données. Le passage par un émulateur affecte les performances en ajoutant de la latence due à ces copies. C'est pourquoi on constate que *Pcl* est plus performant que *Vcl* sur le benchmark GC, qui nécessite beaucoup de petites communications et est donc fortement affecté par la latence.

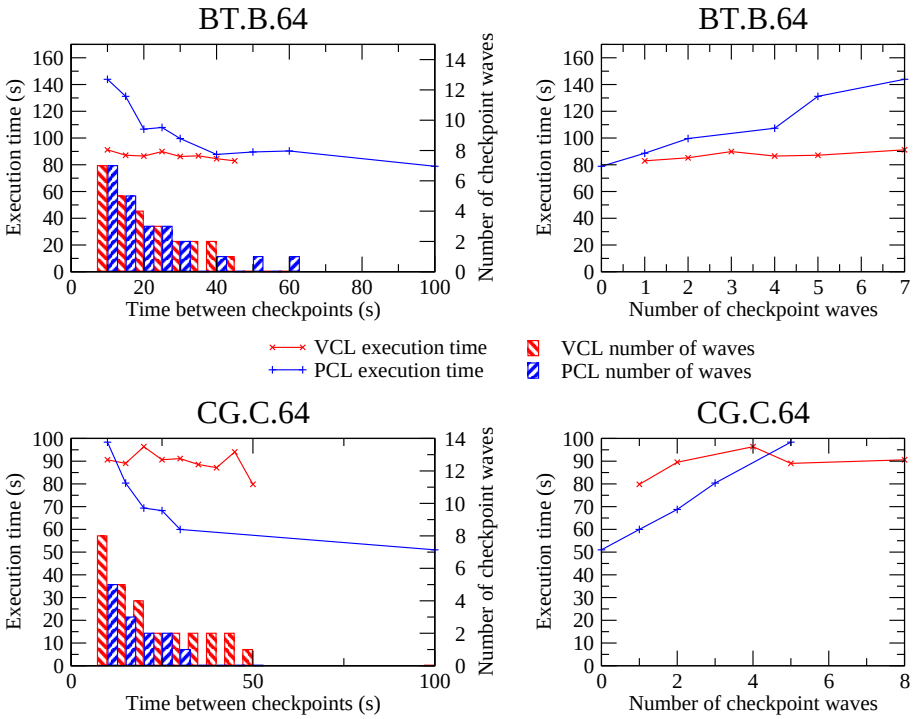


FIG. 19 – Impact de la fréquence des vagues de checkpoints sur réseau myri2000

Le benchmark BT nécessite peu de communications, le latence ajoutée par l'émulation est donc moins importante. On constate alors que *Vcl* présente de meilleures performances pour des fréquences de checkpoints élevées, comme nous l'avions constaté avec les expérience sur grappes de PC.

Conclusion

Cette implémentation nous a permis de comparer une implémentation bloquante à une implémentation non-bloquante de l'algorithme de Chandy-Lamport à travers trois utilisations typiques dans un contexte de calcul à hautes performances : sur cluster de PC, sur cluster à communications à hautes performances, et enfin sur grille de calcul (à grande échelle). Nous avons pu constater le coût des synchronisations de l'implémentation bloquante, notamment sur grille et sur cluster de PC. Les expérience sur les clusters ont montré que pour les deux protocoles, la fréquence des checkpoints est importante, plus que le nombre de processus impliqués. Enfin, les expériences ont montré qu'à des fréquences de checkpoints usuelles, le protocole bloquant a les meilleures performances.

5 MPICH-*Vcausal* pour grilles

5.1 Problématique

On a vu dans la section 2.4.2 l'importance d'un event-logger pour les protocoles de message-logging causal. Cependant, le fait de n'en avoir qu'un seul pour tous les noeuds de calcul est un obstacle à la scalabilité de ces protocoles. De plus, un event-logger unique implique qu'il soit situé sur une machine *stable*, ou considérée comme telle par comparaison avec les autres machines.

Il est donc nécessaire de disposer de plusieurs event loggers à partir du moment où beaucoup de noeuds sont impliqués dans l'exécution d'une application parallèle et/ou lorsqu'il est impossible de garantir qu'une machine sera stable au cours de l'exécution.

5.2 Travaux antérieurs

Dans [6], Lorenzo Alvisi propose une architecture hiérarchique d'event-loggers afin de faire passer à l'échelle le message-logging causal.

Les noeuds de calcul sont rattachés à un event-logger de proximité (ou *proxy*), plusieurs event-loggers de proximité sont rattachés à un event-logger de niveau supérieur, et ainsi de suite.

Les event-loggers assurent une fonction de routage des messages vers des noeuds rattachés à un autre event-logger. Une illustration de cette architecture est visible figure 20. Le processus 2 envoie un message à 3. Tous les deux dépendent de l'event-logger de proximité 1, donc 2 envoie son message directement à 3. Le processus 1 envoie un message au processus 4, qui ne dépend pas du même event-logger de proximité que lui. Il passe donc par son proxy. Comme les proxys des processus 1 et 4 dépendent du même event-logger de niveau supérieur, le proxy de 1 envoie le message au celui de 4, qui le transmet au processus destinataire. Le processus 6, lui, doit envoyer un message à un processus qui n'est pas rattaché à un proxy dépendant du proxy 3. Il envoie son message à son proxy, qui le transmet à son event-logger de niveau supérieur, et ainsi de suite jusqu'à atteindre un event-logger ayant le processus destinataire parmi ses "descendants".

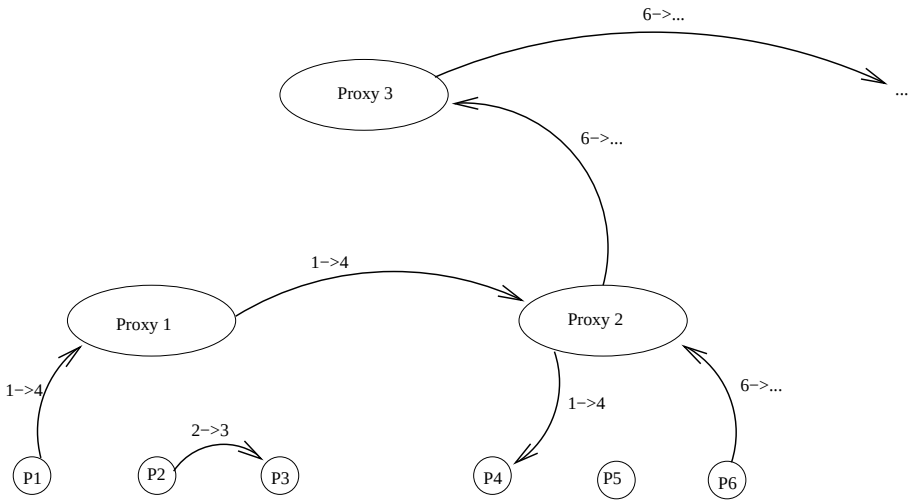


FIG. 20 – Architecture hiérarchique d'event-loggers

Chaque event-logger fait remonter vers son event-logger de niveau supérieur les messages qu'il ne peut pas transmettre lui-même. On risque donc une concentration de l'effort vers les niveaux hiérarchiques élevés. De plus, ce protocole n'est pas tolérant aux pannes au niveau des event-loggers.

5.3 Protocole

5.3.1 La Piecewise Deterministic assumption sur une grille

On se propose ici d'éviter cette hiérarchie, et d'établir un protocole à un seul niveau d'event-loggers.

De la même manière que dans le protocole présenté section 5.2, un ensemble de noeuds de calcul est rattaché à un event-logger donné. Les event-loggers communiquent entre eux, mais deux noeuds rattachés à deux event-loggers différents ne communiquent pas directement entre eux. Comme se place dans un contexte de grille, considérons que l'on dispose d'un event-logger par cluster. Cette architecture est illustrée figure 21. Il est bien sûr possible d'adapter une telle architecture à un cluster de grande taille en séparant en deux l'ensemble des machines et en rattachant chaque partie à un event-logger.

Les informations de causalité concernant les messages échangés au sein de l'ensemble de noeuds dépendant d'un event-logger donné sont stockées sur cet event-logger.

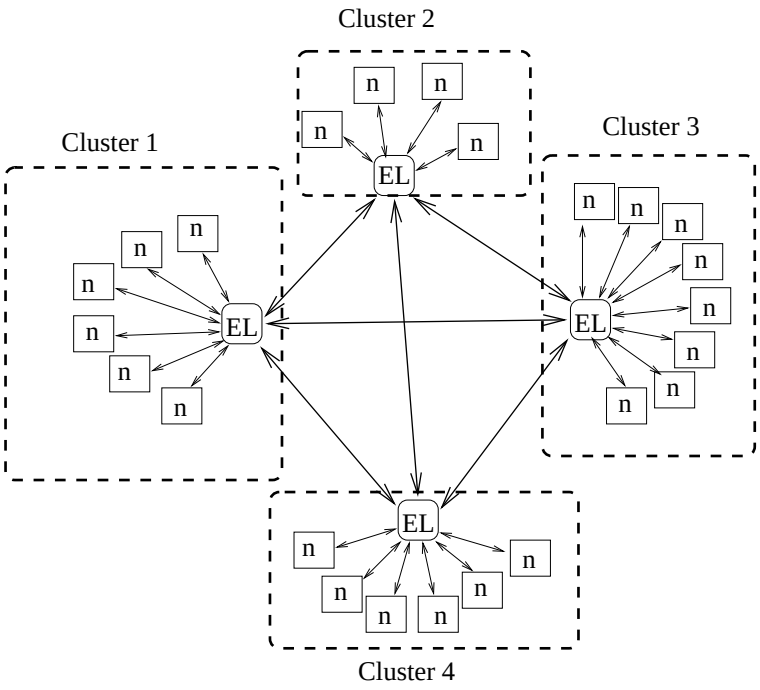


FIG. 21 – Topologie d'une grille typique

Considérons maintenant la *Piecewise Deterministic assumption*, présentée section 2.4.2, à l'échelle de la grille en prenant les clusters comme unité. Lors de l'exécution d'une application parallèle, on a des messages échangés à l'intérieur des clusters et des messages échangés entre les clusters. Ces messages sont des évènements non-déterministes. L'exécution de l'application dans son ensemble est conditionnée par la succession des évènements non-déterministes. Chaque cluster n'agit sur le reste du système qu'avec les messages inter-clusters. Cependant, les informations de causalité entre ces messages inter-clusters sont insuffisant pour assurer la cohérence d'un état de recouvrement après un retour en arrière d'un cluster entier. En effet, considérons un processus A qui reçoit deux messages dans un ordre quelconque, et utilise une valeur contenue dans le dernier message reçu. Si lors de la première exécution, A reçoit d'abord le message d'un processus B (contenant la valeur 0), puis celui d'un processus C (contenant la valeur 2), c'est la valeur 2 qui va être utilisée. Les processus A, B et C effectuent un retour en arrière, et A reçoit maintenant le message de C en premier et celui de B en dernier : c'est la valeur 0 qui sera utilisée. Donc l'ordre des messages intra-cluster est ici également important.

On considère ici l'éventualité d'une panne de la machine sur laquelle est

situé l'event-logger, voir la perte d'un cluster entier. Il est alors nécessaire de répliquer l'information détenue par un event-logger à l'extérieur du cluster.

Les communications au sein d'une grille diffèrent selon si elles ont lieu à l'intérieur d'un cluster ou d'un cluster à un autre. En effet, les réseaux équipant les clusters ont des latences relativement faibles tandis que les clusters sont reliés entre eux par un réseau de type Internet, souvent à très haut débit mais à latence élevée. On a donc intérêt à envoyer des messages courts mais fréquents à l'intérieur d'un cluster et plus gros mais moins fréquents entre les clusters. Le tableau 14 est une comparaison des performances en débit et en latence entre les clusters de Grid5000 et à l'intérieur des clusters. On constate que le réseau est 02 fois plus rapide à l'intérieur des clusters qu'entre deux clusters, et que la latence y est inférieure de deux ordres de grandeur.

Chaque event-logger utilise donc la liaison entre les clusters, peu utilisée, pour envoyer des mises à jour des informations de causalité dont il dispose auprès des autres event-loggers.

Lorsqu'un event-logger effectue une opération non-déterministe, *i.e.,* lorsqu'il envoie un message vers le reste du système, le reste du système doit être à jour des informations de causalité dont cet event-logger dispose. En effet, si la machine où est situé cet event-logger tombe en panne, ou si on perd un cluster tout entier, le reste du système doit avoir connaissance de l'ordre des messages intra et inter-clusters pour les rejouer *jusqu'à la dernière action de ce cluster sur le reste du système.*

On effectue donc des mises à jour à chaque message envoyé vers le reste du système. Si le cluster tombe en panne après le dernier message échangé et que le reste du système n'est pas à jour, cela n'a pas d'importance puisqu'il n'a pas eu d'influence sur le reste du système.

5.3.2 Algorithmes

La signalisation des messages échangés est faite par des envois d'en-têtes.

Un en-tête est composé de trois champs :
- type : signale s'il s'agit d'un message, d'un déterminant, d'un acquittement, d'un message de fin ou d'un avis de checkpoint.
- size : précise la taille de ce qui suit, dans le cas d'un message il s'agit de la taille totale, incluant les déterminants contenus en queue du message.

- rank : donne le rang du destinataire.

Dans le cas d'un acquittement, d'un avis de checkpoint ou d'un message de fin, seul l'en-tête est envoyé. Pour un acquittement ou d'un avis de checkpoint, la valeur de l'horloge logique du processus au moment du déterminant acquitté ou du checkpoint prend la place de la taille.

En-dehors de ces cas, on envoie ensuite un déterminant. S'il s'agit d'un envoi simple de déterminant (d'un client vers un event logger), il s'agit du déterminant que l'event logger doit enregistrer. S'il s'agit d'un message, il s'agit du déterminant qui sera envoyé par le destinataire à l'event logger en remplissant le champ correspondant à l'horloge logique du destinataire au moment de la réception. À la place, ce champ correspond dans un premier temps à la taille du message lui-même, permettant ainsi au destinataire de séparer le message lui-même des déterminants envoyés en queue de message.

Algorithme du client

Envoi

- Si (Envoi d'un message)
  - Si le destinataire fait partie du même cluster :
    - Envoyer le message directement au destinataire
  - Sinon :
    - Envoyer le message à l'event-logger
  - Envoyer l'en-tête :
    - head.type=MESS
    - head.size=taille totale du message avec le piggyback
    - head.rank=rang du destinataire
  - Envoyer un déterminant de signalisation :
    - id_sender=son rang
    - H_r=taille du message seul
    - H_s=son horloge locale
    - nb_prob=0
  - Envoyer le message
  - Envoyer le piggyback
- Si (Envoi d'un déterminant)
  - Envoyer l'en-tête :
    - head.type=DET
    - head.size=taille d'un déterminant
    - head.rank=son rang
  - Envoyer le déterminant

- Si (Envoi d'un avis de checkpoint)
  - Envoyer l'en-tête :
    - head.type=CKPT
    - head.size=valeur de l'horloge locale au moment du checkpoint
    - head.rank=son rang

Réception
- Recevoir l'en-tête
- S'il s'agit d'un acquittement :
  - Recevoir l'acquittement
  - Vider la liste de déterminants qu'on envoie en piggyback
- S'il s'agit d'un message :
  - Recevoir le message
  - recevoir le piggyback
  - Ajouter les déterminants contenus en piggyback à la liste locale

Algorithme de l'event-logger

Gestion des clients
- Recevoir l'en-tête
- S'il s'agit d'un déterminant :
  - Ajouter le déterminant à la table des déterminants
  - Envoyer un acquittement au client
- S'il s'agit d'un message :
  - Recevoir le message
  - recevoir le piggyback
  - Ajouter les déterminants contenus en piggyback à la table des déterminants
  - Envoyer une mise à jour de la table des déterminants au destinataire du message
  - Transmettre le message à son destinataire
- S'il s'agit d'un avis de checkpoint :
  - Retirer de la table des déterminants tous ceux envoyés par le noeud qui a effectué le checkpoint et antérieurs au checkpoint

Gestion des autres event-loggers
- Recevoir un update
- Mettre à jour la table des déterminants en y ajoutant ceux contenus dans l'update
- Si un message suit l'update :

- Recevoir le message
- Transférer l'en-tête du message à son destinataire
- Transférer le message à son destinataire

5.4 Implémentation

L'objectif majeur étant la scalabilité, l'implémentation de l'event logger doit en tenir compte et utiliser le moins possible de ressources par client.

Gestion des connexions

L'event logger est un serveur du point de vue de ses clients, et un client ou un serveur de celui des autres event loggers. Au lancement de l'event logger, on lui passe en paramètre le nombre total d'event loggers du système et les adresses de ceux auxquels il doit se connecter. Ainsi, il se connecte sur ceux dont il a l'adresse (il est donc alors client) et accepte la connexion des autres (il est donc alors serveur).

Dans son fonctionnement normal (en-dehors de la phase d'initialisation), il doit écouter ce qui vient des clients et des autres event loggers. Les sockets sont multiplexées dans l'appel système `poll()`, qui permet d'écouter les évènements sur un tableau de sockets.

Gestion des évènements

Les évènements sur les sockets sont gérés par des threads spécialisés : il y a un type de threads pour gérer les évènements venant des clients et un autre pour les évènements venant des autres event loggers.

En fonctionnement normal, les évènements arrivant des clients sont les suivants :

- Envoi d'un déterminant : l'event logger l'ajoute à sa table des déterminants et envoie un acquittement au client qui l'a envoyé
- Envoi d'un message à destination d'un autre cluster : l'event logger ajoute les informations contenues en queue du message à sa table des déterminants, envoie une mise à jour de la table des déterminants à l'event logger du cluster destinataire et transmet le message à cet event logger
- Avis de checkpoint : l'event logger efface dans sa table des déterminants ceux qui sont antérieurs au checkpoint

Les communications entre les event loggers sont des envois de mises à jours de la table des déterminants et des envois de messages inter-clusters. L'event

logger reçoit d'abord une mise à jour de la table des déterminants puis éventuellement un message.

Les évènements sont gérés par des threads. Quand un évènement survient sur une socket, le thread principal (celui qui écoute sur l'ensemble des sockets avec le `poll()` le transmet à un thread. Le nombre de threads est paramétrable dans le code de l'event logger. En l'absence d'évènements, il faut absolument éviter l'attente active des thread : en effet, l'attente active utilise inutilement du temps CPU en laissant le processus dans la file de l'ordonnanceur des processus. On préfère alors endormir les threads inactifs, qui ne prennent alors pas part à l'ordonnancement général des processus. C'est pourquoi on a choisi ici d'implémenter un algorithme d'ordonnancement patron-ouvrier-secrétaire : le patron réveille les ouvriers pour leur donner du travail, et le secrétaire prévient le patron lorsqu'un certain nombre d'ouvriers ont terminé leur travail. Ce n'est pas un algorithme patron-ouvrier classique en raison de l'utilisation de l'appel système `poll()`.

Algorithmes de gestion des clients

Patron

- Attendre les évènements réseau
- Pour chaque évènement réseau :
  - Créer une entrée dans les jobs et réveiller (éventuellement) un ouvrier
- Supprimer la socket des évènements écoutés
- Signaler au secrétaire qu'il doit réécouter les sockets

Ouvrier

- Supprimer un job de la file
- Pour un job :
  - Lire la donnée du réseau
  - Traiter le job
  - Remettre la socket en écoute pour le patron
  - Prévenir le secrétaire

Secrétaire

- Si (TIMEOUT et nb_de_sockets_en_attente > 0) ou nb_de_sockets_en_attente $>$ SEUIL

– Activer le patron

Les threads ouvriers attendent un signal qui doit venir du patron. Lorsqu'un nouveau job arrive, le patron envoie ce signal : si le thread destinataire de ce signal l'attendait il se réveille, s'il était en train de faire autre chose le signal est perdu. C'est pourquoi lorsque les threads ouvriers ont terminé de traiter leur job, ils regardent s'il n'y en a pas d'autres en file d'attente pour savoir s'ils doivent s'endormir ou traiter un nouveau job.

Les sockets multiplexées dans le `poll()` attendent des évènements en réception (flag `POLLIN`). Lorsqu'un job est traité, la socket n'est plus disponible en réception, et on la fait écouter à nouveau une fois le job traité. Cependant, sous Linux, la structure sur laquelle `poll()` écoute n'est pas modifiable tant qu'on ne sort pas du `poll()` [2]. C'est pourquoi on a besoin du secrétaire, qui fait sortir du `poll()` en provoquant un évènement sur une socket Unix (elle aussi dans le `poll()`) pour le faire rentrer à nouveau dans le `poll()`, mais en écoutant sur la nouvelle structure.

Le secrétaire fait sortir et rentrer le patron dans le `poll()` lorsqu'un certain nombres de sockets doivent être remises en écoute ou lorsqu'un certain temps s'est écoulé sans qu'aucun nouveau signal émis par un client ne lui soit arrivé (timeout).

5.5 Performances

5.5.1 Démarche expérimentale

Le but est ici de tester la scalabilité de ce protocole. Si la perte de temps de calcul en cas de retour arrière est relativement prévisible (dépend principalement de la fréquence des checkpoints), l'overhead induit par le protocole de tolérance aux pannes doit être étudié expérimentalement.

Dans le cadre de ce stage, j'ai réalisé la *conception* et l'*évaluation* d'un protocole de tolérance aux pannes. Il s'agit d'une étude expérimentale située en amont du développement d'une application tolérante aux pannes.

[2] Sous Solaris il existe un pseudo-device /dev/poll sur lequel il est possible d'effectuer les opérations `read()` et `write()`, et ainsi de modifier la structure sur laquelle `poll()` écoute sans en sortir. Il existe un patch pour le noyau Linux permettant de disposer de ce pseudo-device comme sous Solaris, cependant dans un soucis de compatibilité on ne l'utilise pas ici

La phase d'évaluation permet de se rendre compte des performances du protocole, avant éventuellement de passer à la phase de développement d'une librairie MPI tolérante aux fautes utilisant ce protocole. Le coût impliqué par l'évaluation prototypaire d'un protocole étant moindre que celui impliqué par une implémentation complète, si le protocole se révèle inefficace et doit être abandonné, la perte engendrée est moins importante.

L'event logger expérimental

J'ai donc implémenté un event logger qui, à peu de choses près, peut être utilisé dans une implémentation complète. Les détails à modifier concernent les spécificités de l'architecture de la bibliothèque hôte. Par exemple, chaque event logger écoute sur des ports fixés en dur dans le code. Dans une implémentation dans une bibliothèque MPI, ces ports seraient flottants et communiqués à une entité du système, qui se chargerait de les communiquer aux autres entités devant interagir avec les event loggers.

Le client expérimental

Pour tester les performances de cet event logger, j'ai implémenté un client expérimental. Il s'agit d'un pseudo-client qui envoie des messages vers les autres clients, tout en suivant le protocole implémenté. Afin de tester ce protocole dans des conditions les plus réalistes possibles, les messages envoyés sont ceux qui seraient typiquement envoyés entre des processus MPI.

Chaque client parcourt un fichier où il lit les informations sur les messages à envoyer. Trois informations lui sont nécessaires pour jouer des messages réalistes :

- Le rang du destinataire
- La taille du message
- Le temps à attendre avant d'envoyer le message

Cette dernière information permet de simuler le temps de calcul du processus entre deux envois de messages.

Benchmarks

Un bon moyen d'évaluer les performances d'un protocole est de le soumettre à une utilisation typique *via* un benchmark. On fait alors jouer aux pseudo-clients les messages échangés lors de l'exécution d'un benchmark par une bibliothèque MPI non tolérante aux fautes. Ici, on utilise MPICH2.

Avec l'aide d'un membre du projet MPICH-V qui connaît le code de MPICH2, j'ai instrumenté le driver *SOCK* afin d'écrire les informations dé-

crites au paragraphe précédent dans un fichier à chaque fois qu'un paquet est envoyé entre deux clients. Nous avons cherché à nous placer au niveau d'abstraction le plus bas dans l'architecture du driver. En effet, MPICH2 découpe certaines *requêtes* de messages à envoyer entre deux processus MPI en *paquets*. La tolérance aux pannes doit être implémentée au plus bas niveau, de plus le pseudo-client doit envoyer des messages sur le réseau de la même manière qu'un client réel. On s'intéresse donc aux paquets envoyés.

Seuls les envois sont imposés. En effet, si l'on impose les réceptions (en particulier, le moment précis où elles doivent être effectuées), on force trop de paramètres dans le timing des passages de messages du système. L'état du réseau, et en particulier sa charge, ne sont pas les mêmes lors d'une exécution utilisant une bibliothèque MPI "traditionnelle" et lors de l'expérimentation d'un protocole de tolérance aux pannes. Forcer le moment des réceptions induirait alors une latence supplémentaire dans l'exécution du prototype. Cependant, il se peut que l'ordre causal entre les messages ne soit plus respecté, par exemple si un envoi a lieu avant une réception devant normalement le précéder. Encore une fois, il s'agit ici d'un prototype et ce genre d'inexactitudes, s'il y a peu de chances qu'il modifie de manière significative les résultats, justifie des expérimentations avec une implémentation complète du protocole afin de l'évaluer de manière plus fine.

Lancement de la pseudo-application

Afin de démarrer les event loggers et de lancer les clients, j'ai écrit un petit programme qui, exécuté sur une machine, lance les processus distants devant s'exécuter sur les autres machines. À la manière du *mpiexec* de MPICH2, le lanceur parcourt un fichier de machines et lance les event loggers puis les clients correspondants. Pour rendre ce lancement à distance possible, il est nécessaire que ces processus soient des démons. La première chose qu'ils font donc, à leur lancement, est de fermer les entrées et sorties standards (`stdin`, `stdout` et `stderr`), de se dupliquer et de tuer le processus père. Le processus fils est alors démon.

Le fichier de machines diffère légèrement d'un *machinefile* habituel utilisé avec une librairie MPI non tolérante aux fautes. En effet, il faut préciser les machines sur lesquelles les event loggers doivent être lancés, et les machines sur lesquelles les processus clients dépendant d'un event logger sont lancés. Les machines correspondant aux event loggers sont précédées du signe $ et toutes les machines suivant la déclaration d'un event logger en dépendant, jusqu'à la déclaration d'un nouvel event logger. Ainsi, un fichier machine

ressemble à ce qui suit :

```
$ <EL_0>
<client_0 de l'EL_0>
<client_1 de l'EL_0>
<client_2 de l'EL_0>
<client_3 de l'EL_0>
<client_4 de l'EL_0>
...
$ <EL_1>
<client_0 de l'EL_1>
<client_1 de l'EL_1>
<client_2 de l'EL_1>
<client_3 de l'EL_1>
<client_4 de l'EL_1>
...
```

Fin de l'exécution

Chaque client sait qu'il a fini ses envois quand il arrive à la fin du fichier, cependant il n'a peut-être pas effectué toutes ses réceptions. Pour signifier la fin des envois aux autres clients, on envoie un message spécifique END. Dans l'hypothèse où les canaux de communication ont la propriété FIFO, ce message arrive après tous les messages. Chaque client envoie ce message aux autres clients de son cluster, et compte le nombre de clients qui ont terminé leurs envois. Une fois que tous les clients d'un cluster ont terminé, ils envoient un message END à l'event-logger, qui envoie un message END aux autres event-loggers. Lorsque tous les clusters ont terminé, les event-loggers envoient un message END à leurs clients, qui savent ainsi que toute l'exécution est terminé.

En résumé, les messages END sont envoyés dans l'ordre suivant :

- Entre les clients d'un même cluster → une fois qu'il a reçu des messages END provenant de tous les clients de son cluster, un client sait que son cluster a terminé.
- Entre un client et son event-logger (EL) → une fois que le cluster a terminé, l'EL a reçu des messages END de tous ses clients.
- Entre les EL → quand son cluster a terminé, un EL envoie aux autres EL un message END.
- Entre un EL et ses clients → une fois que tous les clusters ont terminé (et que tous les EL ont envoyé leur message END aux autres EL), chaque EL envoie à ses clients le message END.

La propriété FIFO du protocole de transport sous-jacent assure que tous les messages END ne sont reçus qu'une fois que tous les messages précédents ont été réçus. Ainsi, si un client finit ses envois avant d'avoir reçu tous les messages qui lui sont envoyés, il ne termine ses réceptions qu'une fois qu'il a reçu le dernier message END, *i.e.,* celui venant de l'event-logger.

Mesure de performances

On cherche à connaitre la durée de l'exécution de l'application expérimentale. Or, comme dans toute application distribuée, il est impossible d'utiliser un chronomètre global. Deux possibilités s'offrent à nous :

- On peut mesurer la durée entre le moment où le premier event-logger démarre (celui qui sert de serveur à tous les autres event-loggers) et le moment où son exécution se termine. L'avantage de cette mesure est qu'elle rend compte de la durée *totale* de l'exécution de l'application, y compris après que tous les messages aient été joués.
- On peut aussi mesurer la durée entre le moment où les clients commencent leur exécution et celui où ils reçoivent le dernier message END. On cnsidère ensuite comme le temps d'exécution de l'application le plus grand des temps des clients.

C'est cette seconde possibilité qui a été choisie. On place donc des chronomètres dans le client expérimental, et on choisit la plus grande valeur.

5.5.2 Conditions d'expérimentations

J'ai évalué ce protocole en utilisant la plateforme sim du LRI pour les expériences de cluster, et Grid'5000 pour la dernière famille d'expérience. La plate-forme sim est un cluster de 96 machines équipées de processeurs AMD Athlon cadencés à 1,5 GHz connectés par un réseau Gigabit Ethernet. Chaque machine est équipée d'un disque dur IDE d'une capacité de 40 Go. J'ai utilisé le benchmark BT de la suite NAS évoquée dans la section 4. Sur Grid'5000 j'ai utilisé les clusters d'Orsay et de Sophia-Antipolis. Les machines de ces deux clusters sont équipées de processeurs AMD Opteron connectés par un réseau Gigabit Ethernet. La connexion entre les deux clusters est assurée apr le fournisseur Renater au moyen d'une Fibre Noire. Les performances des communications sur Grid'5000 sont présentés figure 14.

5.5.3 Résultats et interprétation

Le but de ce protocole est d'obtenir de meilleures performances en distribuant l'event logger. Il est donc intéressant d'observer l'évolution du temps d'exécution d'une application en fonction du nombre d'event loggers utilisés.

J'ai tout d'abord utilisé le benchmark W, qui manipule des matrices de petite taille. Les résultats avec 49 noeuds de calcul sont présentés figure 22. On constate que le temps d'exécution ne varie pas de manière significative quel que soit le nombre d'event loggers utilisé : à cette échelle, l'event logger n'est pas un élément critique. Le fait d'utiliser un event logger distribué n'améliore pas les performances, mais il ne les diminue pas non plus : on peut au moins s'en servir pour tolérer les pannes sur ce composant en le répliquant.

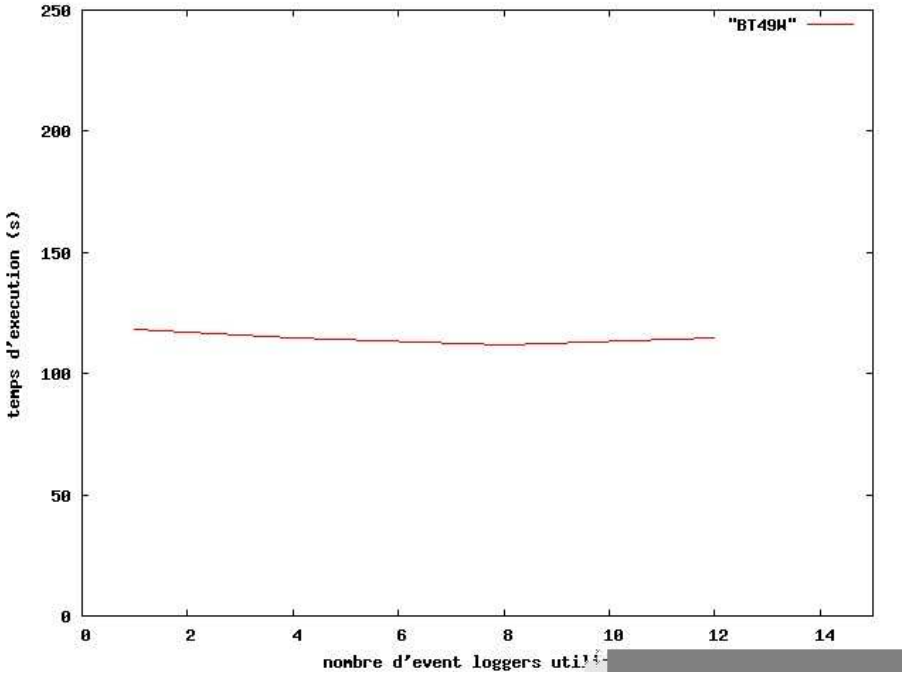


FIG. 22 – Exécution sur 49 noeuds de calcul

À plus grande échelle et sur un benchmark plus long, on constate figure 23 une diminution du temps d'exécution sur la première partie de la courbe. L'event logger distribué améliore donc effectivement les performances du protocole. Si l'on utilise moins de 8 event loggers, les performances sont améliorées, puis elles ne varient plus de manière significative si l'on utilise plus d'event loggers. À partir de 8 event loggers, pour un calcul de cette taille, l'event logger n'est plus un élément ralentissant de l'application.

La dernière famille d'expériences a pour but de mesurer les performances du protocole sur une grille. On utilise un benchmark BT de classe A s'exécutant sur 25 processus. À cette échelle, on a vu que le nombre d'event loggers

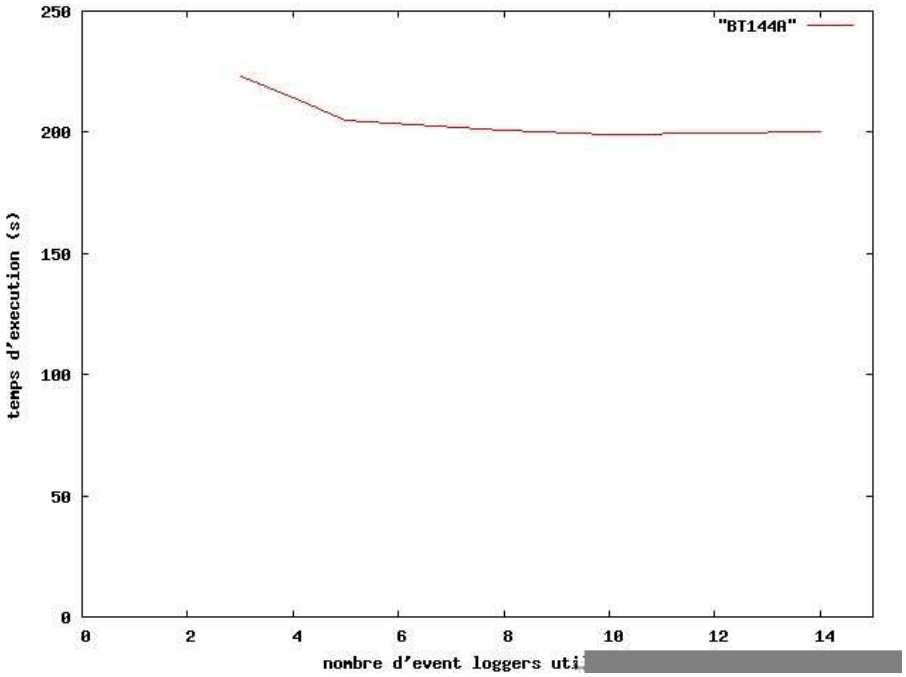


FIG. 23 – Exécution sur 144 noeuds de calcul

n'a pas d'influence sur les performances. L'intérêt ici est de voir comment le protocole s'adapte à la topologie de la grille. J'ai donc fait varier la proportion de processus s'exécutant dans chacun des clusters : d'abord, tous les processus sont situés à Orsay, puis on en place de plus en plus à Sophia-Antipolis, jusqu'à ce que les processus soient répartis quasi-équitablement sur les deux clusters (13 et 12 processus). Deux séries de mesures sont effectuées : la première en n'utilisant qu'un seul event logger, situé à Orsay, la deuxième en utilisant un event logger à Orsay et un autre à Sophia-Antipolis. Les temps d'exécution sont représentés figure 24.

Lorsque les processus sont tous situés à Orsay, les performances sont identiques. L'event logger de Sophia-Antipolis n'est pas utilisé, donc les deux configurations avec un et deux event loggers ont une exéuction identique. Lorsque l'on situe des processus à Sophia-Antipolis, les performances se dégradent du fait des temps de communications entre les processus. La courbe rouge de la figure 24 représente le temps d'exécution en n'utilisant qu'un seul event logger, situé à Orsay. Plus on utilise le processus à Sophia-Antipolis, plus l'exécution est ralentie, les processus de Sophia-Antipolis devant faire appel à un event logger situé à Orsay. La courbe verte représente le temps d'exécution en utilisant un event logger dans chaque cluster : les processus

situés à Orsay font appel à l'event logger situé à Orsay, ceux situés à Sophia-Antipolis font appel à celui situés à Sophia-Antipolis. Chaque processus fait appel à un event logger local. Le ralentissement est donc moins important en utilisant deux event loggers locaux.

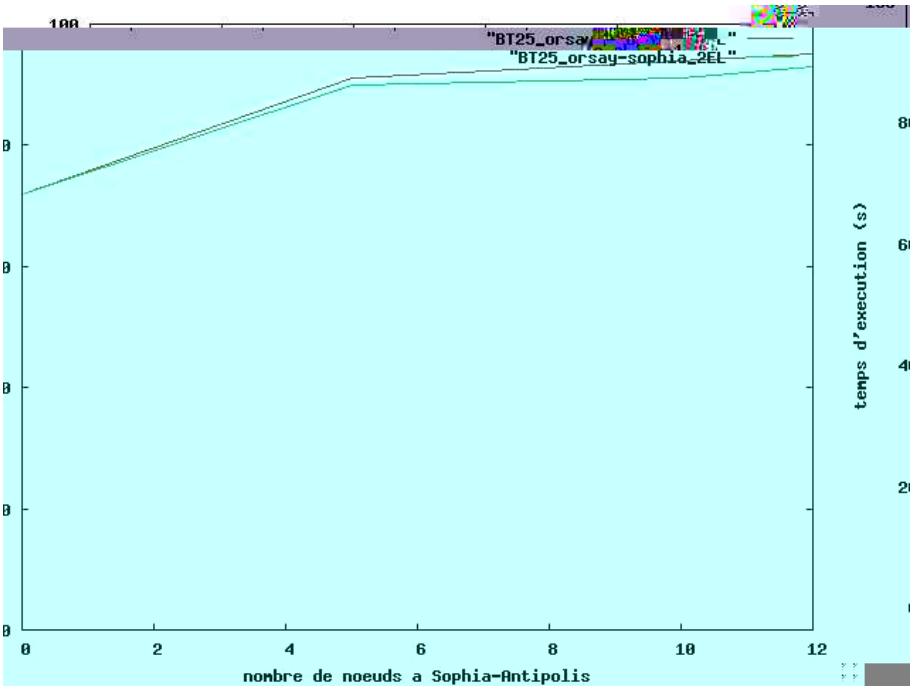


FIG. 24 – Exécution sur deux clusters

6 Conclusion

Durant ce stage j'ai participé à l'évaluation de deux protocoles de tolérance aux défaillances dans les applications parallèles communiquant par passage de messages.

Dans un premier temps, j'ai pris part à l'implémentation de MPICH2-*Pcl*, une implémentation bloquante de l'algorithme de Chandy-Lamport dans MPICH2. Il s'agit d'une technique de sauvegarde de l'état global d'une application dans un but de retour en arirère de l'exécution en cas de panne. Nous avons évalué ce protocole dans trois contextes de calcul à hautes performances (grille de calcul, cluster, cluster à communications à hautes performances). Les expériences ont montré que les synchronisations entre les noeuds de calcul induites par ce protocole pénalisent davantage l'exécution du calcul qu'avec un protocole non-bloquant. Cependant, les expériences ont montré qu'en utilisant des fréquences de prises de checkpoints usuelles les performances de ce protocole sont meilleures sur un cluster à communications à hautes performances.

La deuxième partie de ce stage a été consacrée à la conception et l'évaluation d'un protocole hiérarchique de tolérance aux pannes par checkpoints non-coordonnés basé sur un mécanisme d'enregistrement des messages. Il s'agissait de rendre possible le passage à l'échelle d'un protocole impliquant un enregistrement sur un support distant des informations de causalité entre les messages. J'ai donc proposé un protocole hiérarchique basé sur un enregistreur d'évènements distribué. Les expériences ont montré que la distribution de ce composant décroît les temps d'exécution.

L'augmentation de la taille des systèmes de calcul à hautes performances rend encore plus indispensable la tolérance aux défaillances. Si les protocoles actuels sont relativement bien maîtrisés dans les clusters, ils ne s'avèrent pas aussi efficaces dans les grilles de calcul. C'est pourquoi on cherche aujourd'hui à développer des protocoles spécifiques aux grilles et non plus à utiliser directement ceux qui ont fait leurs preuves dans les clusters. La première question qui se pose est la manière d'effectuer des communications efficaces dans les grilles, compte tenu de l'hétérogénéité du système. Ensuite, on cherche à développer des protocoles de tolérance aux pannes efficaces dans les grilles en étudiant les solutions existantes puis en en proposant de nouvelles comme les protocoles hiérarchiques.

Références

[1] Top500. http ://www.top500.org.

[2] A. Agbaria and R. Friedman. Starfish : Fault-tolerant dynamic MPI programs on clusters of workstations. In *8th International Symposium on High Performance Distributed Computing (HPDC-8 '99)*. IEEE CS Press, August 1999.

[3] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An analysis of communication induced checkpointing. In *29th Symposium on Fault-Tolerant Computing (FTCS'99)*. IEEE CS Press, june 1999.

[4] Lorenzo Alvisi and Keith Marzullo. Message logging : Pessimistic, optimistic, causal, and optimal. *IEEE Trans. Software Eng*, 24(2) :149–159, 1998.

[5] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, 1995.

[6] Karan Bhatia, Keith Marzullo, and Lorenzo Alvisi. Scalable causal message logging for wide-area environments. *Concurrency and Computation : Practice and Experience*, 15(10) :873–889, 2003.

[7] Aurélien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2 : a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *High Performance Networking and Computing (SC2003)*. Phoenix USA, IEEE/ACM, November 2003.

[8] Aurélien Bouteiller, Pierre Lemarinier, Géraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. In *IEEE International Conference on Cluster Computing (Cluster 2003)*. IEEE CS Press, December 2003.

[9] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *PPOPP*, pages 84–94. ACM, 2003.

[10] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in application-level fault-tolerant mpi. In *ICS '03 : Proceedings of the 17th annual international conference on Supercomputing*, pages 234–243, New York, NY, USA, 2003. ACM Press.

[11] Greg Burns, Raja Daoud, and James Vaigl. LAM : An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[12] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Yvon Jegou, Pascale Vicat-Blanc Primet, Emmanuel Jeannot, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Benjamin Quetier, and Olivier Richard. Grid'5000 : A large scale and highly reconfigurable grid experimental testbed. In *SC'05 : Proc. The 6th IEEE/ACM International Workshop on Grid Computing CD*, pages 99–106, Seattle, Washington, USA, November 2005. IEEE/ACM.

[13] K. M. Chandy and L.Lamport. Distributed snapshots : Determining global states of distributed systems. In *Transactions on Computer Systems*, volume 3(1), pages 63–75. ACM, February 1985.

[14] Yuqun Chen, Kai Li, and James S. Planck. CLIP : A checkpointing tool for message-passing parallel programs. In *SC97 : High Performance Networking and Computing (SC97)*. IEEE/ACM, November 1997.

[15] E. N. Elnozahy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Symposium on Reliable Distributed Systems*, pages 39–47, 1992.

[16] E. N. Elnozahy and W. Zwaenepoel. Replicated distributed processes in manetho. In *22nd International Symposium on Fault Tolerant Computing (FTCS-22)*, pages 18–27, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[17] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3) :375 – 408, september 2002.

[18] Graham E. Fagg and Jack Dongarra. FT-MPI : Fault tolerant MPI, supporting dynamic applications in a dynamic world. In Jack Dongarra, Péter Kacsuk, and Norbert Podhorszki, editors, *PVM/MPI*, volume 1908 of *Lecture Notes in Computer Science*, pages 346–353. Springer, 2000.

[19] Graham E. Fagg and Jack J. Dongarra. HARNESS fault tolerant MPI design, usage and performance issues. Technical report ? ? ? ?, University of Tennessee, Knoxville, Knoxville, TN 37996, USA, 2002.

[20] Message Passing Interface Forum. MPI : A message-passing interface standard. Technical Report UT-CS-94-230, Department of Computer Science, University of Tennessee, April 1994. Tue, 22 May 101 17 :44 :55 GMT.

[21] 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.

[22] Al Geist, William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing L. Lusk, William Saphir, Anthony Skjellum, and Marc Snir. MPI-2 : Extending the message-passing interface. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par, Vol. I*, volume 1123 of *Lecture Notes in Computer Science*, pages 128–135. Springer, 1996.

[23] William Gropp and Ewing Lusk. Fault tolerance in MPI programs. *special issue of the Journal High Performance Computing Applications (IJHPCA)*, 2002.

[24] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6) :789–828, September 1996.

[25] Rachid Guerraoui and Andre Schiper. Software based replication for fault tolerance. *IEEE Computer*, 30(4) :68–74, April 1997.

[26] Jean-Michel Hélary, Achour Mostefaoui, and Michel Raynal. Communication-induced determination of consistent snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 10(9) :865–877, 1999.

[27] E. Roman J. Duell, P. Hargrove. The design and implementation of berkeley lab's linux checkpoint/restart. Technical Report publication LBNL-54941, Berkeley Lab, 2003.

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

[29] Pierre Lemarinier, Aurélien Bouteiller, Thomas Herault, Géraud Krawezik, and Franck Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *IEEE International Conference on Cluster Computing (Cluster 2004)*. IEEE CS Press, 2004.

[30] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, 1997.

[31] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2) :220–232, 1975.

[32] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI

checkpoint/restart framework : System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[33] F. B. Schneider. The state machine approach : A tutorial. In *Fault-Tolerant Distributed Computing*, pages 18–41, Berlin - Heidelberg - New York, 1987. Springer.

[34] 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. page 38. IEEE Computer Society, April 26 2004.

[35] Q. Snell, A. Mikler, and J. Gustafson. Netpipe : A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems.*, June 1996.

[36] Georg Stellner. CoCheck : Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, April 1996. IEEE CS Press.

[37] E. Strom and S. Yemini. Optimistic recovery in distributed systems. In *Transactions on Computer Systems*, volume 3(3), pages 204–226. ACM, August 1985.

[38] V.C. Zandy. libckpt, 2005. http ://www.cs.wisc.edu/ zandy/ckpt/.

[39] Xianan Zhang, Flavio Junqueira, Matti Hiltunen, Keith Marzullo, and Richard Schlichting. Replicating nondeterministic services on grid environments. In *HPDC*, pages 105–116, 2006.

Index

Canal de communications, 15
Chandy-Lamport (algorithme), 20
CIC, 28

Effet domino, 20
Etat cohérent, 19
Event logger, 26

Fautes, voir Pannes

Grand Large, 12
Grid'5000, 13

INRIA Futurs, 12

Ligne de recouvrement, 20
LRI, 10

Message logging, 23
    causal, 25
    optimiste, 24
    pessimiste, 25
MPI, 30
MPI-V, 14
MPICH-*Vcausal*, 48
MPICH-*Vcl*, 40
MPICH2*Pcl*, 34

Pannes, 15
    définitives, 16
    franches, 16
    temporaires, 15
Piecewise Deterministic Asumption, 23
Processus, 15
processus orphelins, 24

Réplication, 16
    active, 17
    primaire/sauvegarde, 17
Retour sur points de reprise, 18
    coordonnés, 20
    non coordonnés, 23

Systèmes distribués, 15