Conception d'un modèle et de frameworks de distribution d'applications sur grappes de PCs avec tolérance aux pannes à faible coût

Constantinos Makassikis

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR des Sciences & Technologies

Conception d'un modèle et de frameworks de distribution d'applications sur grappes de PCs avec tolérance aux pannes à faible coût

THÈSE

présentée et soutenue publiquement le 2 février 2011

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Constantinos MAKASSIKIS

Composition du jury

Président :	Laurent PHILIPPE	Professeur, Université de Franche Comté
Rapporteurs :	Pierre MANNEBACK	Professeur, Université de Mons
	Serge CHAUMETTE	Professeur, Université Bordeaux I
Examinateurs :	Claude GODART	Professeur, Université de Lorraine
	Stéphane VIALLE	Professeur, Supélec Campus de Metz (directeur de thèse)
	Virginie GALTIER	Professeur adjoint, Supélec Campus de Metz (co-encadrante de la thèse)
	Xavier WARIN	EDF R&D (Équipe Osiris)

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Supélec
ÉCOLE SUPÉRIEURE D'ÉLECTRICITÉ

Mis en page avec la classe thloria.

Remerciements

Que ce soit à Supélec, au LORIA, ou ailleurs, nombreuses sont les personnes qui ont contribué d'une manière ou d'une autre à la réalisation du travail exposé dans ce manuscrit. D'emblée j'aimerais les remercier et demander pardon à toutes celles que je n'ai pas mentionnées ci-après.

Tout d'abord, je tiens à exprimer ma gratitude envers mon directeur de thèse, le professeur Stéphane Vialle, et ma co-encadrante de la thèse, le docteur Virginie Galtier, pour m'avoir guidé formidablement tout au long de ces quatre dernières années. Sans leur perspicacité, sans leur exigence, sans leur rigueur, sans leur disponibilité, sans leur pédagogie et sans leurs conseils, Dieu sait où je me serais embourbé : leur implication active et complémentaire a été instrumentale pour compléter ma formation et pour mener à bien ce projet. Je leur suis aussi très reconnaissant pour avoir partagé leur expérience et m'avoir fait découvrir et apprécier différentes facettes du monde de la recherche et de l'enseignement. À travers nos discussions, je me suis beaucoup enrichi. Stéphane, Virginie, je suis très heureux que nous ayons travaillé ensemble.

Ensuite, je remercie les membres de mon jury de thèse. À commencer par les professeurs Pierre Manneback et Serge Chaumette qui ont accepté de rapporter mon mémoire, puis les professeurs Laurent Philippe et Claude Godart, et pour finir, Xavier Warin d'EDF avec lequel j'ai eu le plaisir de travailler en début de thèse dans le cadre du projet ANR-GCPMF.

Un grand merci également au professeur Jens Gustedt et à toute l'équipe AlGorille pour m'avoir accueilli et soutenu. Votre aide pour la préparation de la soutenance et vos encouragements furent très précieux.

Enfin, je remercie les membres du personnel de Supélec que j'ai côtoyés au cours de ces quatre années et qui ont facilité sinon rendu mon séjour plus agréable. Pour ne citer qu'une partie : Patrick, Claudine, Sébastien, Gillou, merci d'avoir égayé le second étage en instaurant une ambiance toute particulière dont vous détenez le secret. Hervé, merci d'avoir rendu plus pimentées certaines journées avec des blagues et des répliques mémorables telles que « Buenas, burras » ou « Tu perds déjà du temps ».

Je ne pourrai clore cette énumération sans citer Lucian et Matthieu que j'ai beaucoup côtoyés et avec lequels j'ai beaucoup échangé. En particulier, Lucian qui fut mon complice. Je n'oublierai pas les longues heures passées ensemble au bureau à travailler chacun sur nos thèses respectives, ce qui nous a valu le surnom de *stakhanovistes*. Je n'oublierai pas non plus les innombrables parties de billard puis de tennis de table, ainsi que les sorties touristiques, les soirées gastronomiques et tous les autres bons moments passés ensemble.

Avant d'adresser quelques mots dans ma langue maternelle, je tiens à remercier la Région Lorraine et Supélec pour avoir financé ce travail, ainsi que toutes les personnes que je n'ai pas citées.

Τέλως θα ήθελα να αφιερώσω αυτές τις τελευταίες γραμμές στους φίλους μου από το Στρασβούργο, όπως και επίσης, στην οικογένεια μου: Πατέρα, Μητέρα, Μάρθα, Σταματία και Μάριε, σας ευχαριστώ ολόψυχα για την συμπαράστασή σας όλα αυτά τα χρόνια.

If I had more time, I could write a shorter letter.
— Pascal's Law

À ma chère famille.

Résumé

Les grappes de PCs constituent des architectures distribuées dont l'adoption se répand à cause de leur faible coût mais aussi de leur extensibilité en termes de nœuds. Notamment, l'augmentation du nombre des nœuds est à l'origine d'un nombre croissant de pannes par arrêt qui mettent en péril l'exécution d'applications distribuées. L'absence de solutions efficaces et portables confine leur utilisation à des applications non critiques ou sans contraintes de temps.

MoLOToF est un modèle de tolérance aux pannes de niveau applicatif et fondée sur la réalisation de sauvegardes. Pour faciliter l'ajout de la tolérance aux pannes, il propose une structuration de l'application selon des squelettes tolérants aux pannes, ainsi que des collaborations entre le programmeur et le système de tolérance des pannes pour gagner en efficacité.

L'application de *MoLOToF* à des familles d'algorithmes parallèles SPMD et Maître-Travailleur a mené aux frameworks *FT-GReLoSSS* et *ToMaWork* respectivement. Chaque framework fournit des squelettes tolérants aux pannes adaptés aux familles d'algorithmes visées et une mise en œuvre originale. *FT-GReLoSSS* est implanté en C++ au-dessus de MPI alors que *ToMaWork* est implanté en Java au-dessus d'un système de mémoire partagée virtuelle fourni par la technologie JavaSpaces.

L'évaluation des frameworks montre un surcoût en temps de développement raisonnable et des surcoûts en temps d'exécution négligeables en l'absence de tolérance aux pannes. Les expériences menées jusqu'à 256 nœuds sur une grappe de PCs bi-cœurs, démontrent une meilleure efficacité de la solution de tolérance aux pannes de *FT-GReLoSSS* par rapport à des solutions existantes de niveau système (LAM/MPI et DMTCP).

Mots-clés: tolérance aux pannes, points de reprise, squelettes de programmation, algorithmes SPMD, algorithmes Maître-Travailleur, framework.

Abstract

PC clusters are distributed architectures whose adoption spreads as a result of their low cost but also their extensibility in terms of nodes. In particular, the increase in nodes is responsable for the increase of fail-stop failures which jeopardize distributed applications. The absence of efficient and portable solutions limits their use to non critical applications or without time constraints.

MoLOToF is a model for application-level fault tolerance based on checkpointing. To ease the addition of fault tolerance, it proposes to structure applications using fault-tolerant skeletons as well as collaborations between the programmer and the fault tolerance system to gain in efficiency.

The application of *MoLOToF* on SPMD and Master-Worker families of parallel algorithms lead to *FT-GReLoSSS* and *ToMaWork* frameworks respectively. Each framework provides fault-tolerant skeletons suited to targeted families of algorithms and an original implementation. *FT-GReLoSSS* uses C++ on top of MPI while ToMaWork uses Java on top of virtual shared memory system provided by JavaSpaces technology.

The frameworks' evaluation reveals a reasonable time development overhead and negligible runtime overheads in absence of fault tolerance. Experiments up to 256 nodes on a dualcore PC cluster, demonstrate a better efficiency of *FT-GReLoSSS*' fault tolerance solution compared to existing system-level solutions (LAM/MPI and DMTCP).

Keywords: fault tolerance, checkpoints, programming skeleton, SPMD algorithms, Master-Worker algorithms, framework.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

En 1965, Gordon Moore faisait une prédiction sur l'évolution du nombre de transistors par circuit integré au cours du temps [91]. Au vu des évolutions technologiques, cette prédiction, connue sous le nom de « loi de Moore », a été réajustée dix ans plus tard et continue d'être vérifiée depuis. Dans sa formulation la plus récente [92] la loi prédit que le nombre de transistors par circuit integré double tous les deux ans.

Pendant longtemps, l'évolution du nombre de transistors a été accompagnée d'une augmentation de la puissance des processeurs sous forme d'une augmentation de la fréquence d'horloge de ces derniers. Vers 2003, l'augmentation des fréquences des processeurs commence à atteindre ses limites : consommations énergétiques et surchauffes excessives conduisent à repenser l'architecture des processeurs. Dorénavant, les fréquences sont revues à la baisse mais on assiste à une augmentation régulière du nombre de cœurs : d'abord deux, puis quatre, puis six, puis huit. Dernièrement sont apparus des processeurs à douze cœurs [150] et d'autres avec plus de cœurs sont en préparation.

Conjointement à cette évolution, qui se traduit par une augmentation du nombre de cœurs par socket, l'exploitation des nouveaux apports en puissance de calcul continue de suivre le comportement mis en évidence par la « loi de Gustafson » [62]. À savoir que l'utilisateur met à profit le gain en puissance, par exemple, pour améliorer la précision de ses calculs plutôt que pour les accélerer. En d'autres termes, l'utilisateur préfère traiter des problèmes plus gros dans le même intervalle de temps plutôt que de réduire le temps d'exécution du problème courant.

Devant l'augmentation du nombre de nœuds, l'augmentation du nombre de composants par nœud et le maintien des temps d'exécution, l'amélioration du niveau de fiabilité des composants matériels ne suit pas [137]. Conséquemment, la probabilité d'occurrence de pannes augmente également de manière très importante et appelle à la mise en place de solutions efficaces de tolérance aux pannes.

Réaliser de la tolérance aux pannes au niveau applicatif est intéressant car cela permet de profiter de la sémantique applicative afin d'améliorer l'efficacité de la tolérance aux pannes. Par exemple, certains algorithmes parallèles peuvent bénéficier de techniques de tolérance aux pannes spécifiques. Mais implanter des algorithmes parallèles est une tâche généralement complexe et demande un savoir faire. L'ajout de la tolérance aux pannes au niveau applicatif n'est pas une tâche plus aisée et demande des compétences complémentaires en tolérance aux pannes au programmeur (ou développeur).

Notre approche a consisté à regrouper des algorithmes parallèles en familles et à proposer pour chaque famille un framework de manière à masquer ou simplifier l'ajout de la tolérance aux pannes au développeur. Ces frameworks sont construits au-dessus de bibliothèques spécialisées

et demeurent compatibles avec des outils classiques comme MPI, OpenMP, JavaSpaces, etc.

Contributions

Comme nous le disions précédemment, la tolérance aux pannes au niveau applicatif est connue pour être plus efficace que la tolérance aux pannes de niveau système car elle permet d'exploiter des informations sémantiques de l'application. En revanche, elle est également plus difficile à mettre en œuvre. En effet, elle requiert du programmeur des transformations au niveau du code source de son application. Ces transformations complexifient l'application et facilitent l'introduction de bogues. Il en résulte, en général, beaucoup de frustration et des temps de développement plus longs.

Pour pallier ces difficultés, nous introduisons un nouveau modèle de programmation appelé *MoLOToF* qui introduit des squelettes tolérants aux pannes et des collaborations avec le programmeur. Pour faciliter davantage le travail du programmeur, le modèle est associé à deux paradigmes de programmation parallèle très utilisés. À savoir le paradigme *Single Program Multiple Data* (SPMD) et le paradigme *Master-Worker* (MW) (ou Maître-Travailleur). Il en a résulté les frameworks *FT-GReLoSSS* et *ToMaWork* qui sont présentés au chapitre 5 et au chapitre 6 respectivement.

Organisation du manuscrit

Le manuscrit ci-présent est organisé de la manière suivante. Le **chapitre 2** s'intéresse à la distribution large échelle d'une application financière, réalisée en collaboration avec EDF R&D dans le cadre du projet ANR-GCPMF. Elle est représentative d'une classe d'applications qui se caractérisent par des durées d'exécution longues et des dépendances non triviales entre les processus (qui les constituent). Communes dans le milieu académique scientifique, ces applications sont soumises dans le milieu industriel à des échéances plus strictes mais dont le respect peut être compromis par l'occurrence de moins en moins rare de pannes. Les pannes rencontrées au cours de nos expérimentations ont orienté nos recherches vers l'élaboration d'un nouveau modèle pour une tolérance aux pannes à faible coût.

Le **chapitre 3** dresse un état de l'art de la tolérance aux pannes dans les grappes (ou clusters) de PCs. Une première partie est consacrée à différentes définitions relatives aux pannes avant d'aborder, dans une seconde partie, les techniques utilisées en tolérance aux pannes. Les techniques fondées sur une reprise par retour arrière sont les plus répandues pour faire face aux pannes qui surviennent dans les grappes de PCs. Elles reposent sur la capacité à sauver et à restituer les processus de l'application dans un état cohérent. La cohérence est assurée par la présence de protocoles dits « de reprise par retour arrière ». La sauvegarde est assurée par des mécanismes de réalisation de points de reprise. Nous proposons une classification de ces protocoles, suivie de la description d'instances majeures des différentes classes présentées. Nous proposons ensuite une classification des mises en œuvres concrètes de systèmes de réalisation de points de reprise. Ces dernières sont différenciées par la technique utilisée pour réaliser les points de reprise. Enfin, nous terminons en exposant des évolutions récentes visant à rendre plus efficace la tolérance aux pannes.

Le **chapitre 4** présente MoLOToF, le modèle que nous proposons pour le développement d'applications de calculs distribués tolérantes aux pannes. Cette tolérance aux pannes repose sur la réalisation de points de reprise au niveau applicatif tout en cherchant à minimiser les efforts du développeur. Le modèle s'organise autour de *squelettes tolérants aux pannes*. Ces derniers sont proposés par le système de réalisation de points de reprise et il sont utilisés par le programmeur

pour construire son application tolérante aux pannes. MoLOToF met en avant diverses collaborations du système de tolérance aux pannes avec le développeur, mais aussi avec l'environnement d'exécution dans le but de réduire le coût de la tolérance aux pannes.

Les **chapitres 5** et **6** présentent FT-GReLoSSS et ToMaWork. Ce sont deux frameworks issus de l'application du modèle MoLOToF aux paradigmes de parallélisation SPMD et Maître-Travailleur. Conformément au modèle MoLOToF, chaque framework propose des squelettes tolérants aux pannes qui sont adaptés au paradigme de parallélisation consideré et ce, de manière à faciliter l'implantation des applications mais aussi pour obtenir une tolérance aux pannes plus efficace. La flexibilité des frameworks est illustrée au travers d'exemples d'implantation d'applications types.

Le **chapitre 7** concerne l'évaluation des frameworks FT-GReLoSSS et ToMaWork. Les expériences portent d'abord sur la vérification du fonctionnement correct des mécanismes de tolérance aux pannes. Ensuite elles portent sur l'évaluation du niveau de facilité de développement d'applications tolérantes aux pannes avec nos framework. Enfin, elles portent sur l'évaluation de la performance de la tolérance aux pannes.

Le **chapitre 8** conclut le manuscrit en rappellant les principaux résultats et les perspectives ouvertes par notre travail de recherche.

Chapitre 2

Étude de cas : distribution d'un algorithme de contrôle stochastique

Dans ce chapitre nous nous intéressons à la présentation d'un travail que nous avons mené en début de thèse avec l'équipe Osiris d'EDF R&D. Ce travail porte sur la distribution large échelle d'une application financière d'EDF utilisée pour la valorisation d'actif de stockage de gaz et qui s'inscrivait dans le cadre du projet ANR GCPMF. L'objectif de ce projet était d'étudier la faisabilité de l'utilisation de grappes et de grilles de calculs pour permettre l'exécution d'applications financières (valuation de risque de crédits, d'options, etc.). Ces applications sont caractérisées par des besoins en puissance de calcul et en volumes mémoire aux limites des possibilités des machines séquentielles actuelles. Le parallélisme se pose alors comme une solution incontournable pour pallier ces limitations.

De nombreuses applications de calculs financiers se décomposent simplement en tâches indépendantes et donnent lieu à du *bag of tasks*. Cependant, il existe des applications qui ne se prêtent pas à ce type de décomposition. C'est le cas de l'application de valorisation d'actif de stockage de gaz que nous avons parallélisée selon le paradigme de programmation SPMD (ou *Single Program Multiple Data*). La parallélisation met en jeu une décomposition de domaine régulière avec de nombreuses communications, mais variable dans le temps et avec des échanges de frontières irrégulières.

Après une description du contexte financier du calcul de l'application considérée (Section 2.1), nous présentons l'algorithme séquentiel de contrôle stochastique au cœur de l'application financière suivi de sa version parallélisée (Section 2.2). Par la suite (Section 2.3), nous décrivons les expériences menées afin d'évaluer les performances de notre algorithme distribué, et nous partageons nos observations quant aux pannes rencontrées pendant la phase d'expérimentation. Nous concluons (Section 2.4) ce chapitre en rappelant d'abord les résultats clés du travail présenté, puis ensuite en relatant la façon dont il a influencé d'autres travaux, dont les notres en tolérance aux pannes.

2.1 Contexte du calcul financier

Un actif de stockage de gaz est habituellement constitué d'une cavité destinée à contenir le gaz ainsi que de compresseurs permettant d'effectuer des injections, des sous-tirages ou visant simplement à maintenir le gaz sous pression dans la cavité. En temps normal, le propriétaire remplit son actif de stockage pendant les périodes où les prix du gaz sont bas pour répondre aux demandes des clients ou pour le revendre ultérieurement sur les marchés. Éventuellement, il peut

envisager de louer une partie de sa capacité de stockage à une autre entité.

Les prix du gaz connaissent des fluctuations issues principalement de la modification de la demande. À cause de contraintes physiques sur l'extraction et la production de gaz, on observe une inélasticité entre la production et la demande, les prix du gaz sont, par exemple, plus élevés en hiver qu'en été. Le propriétaire d'un actif de stockage de gaz peut alors profiter de cette dynamique des prix en arbitrant entre les différentiels temporels des prix du gaz et ainsi valoriser son actif (c.-à-d. son « réservoir » de gaz). La détermination du prix de location tient compte de l'opportunité d'arbitrage mais elle est également soumise à des contraintes. D'une part, il existe des contraintes physiques liées à la cavité et à la manière de stocker le gaz. Par exemple, une injection (resp. un sous-tirage) est d'autant plus difficile à réaliser (et donc coûteuse) que la cavité est remplie (resp. vide). D'autre part, le contrat de location peut imposer une valeur de stock minimale et le volume loué peut être variable au cours du temps et soumis à des restrictions en ce qui concerne la capacité journalière d'injection (resp. de sous-tirage) autorisée.

La valorisation fait appel à des algorithmes de contrôle stochastique et à des modèles de prix du gaz. De nombreux travaux de recherche récents dans le domaine de la valorisation des actifs de stockage de gaz (cf. [71, 10]) ont conduit à l'élaboration de ces modèles. Cependant, les besoins de ces modèles en termes de puissance de calcul et de consommation mémoire peuvent aisément surpasser les capacités des machines monoprocesseur actuelles ce qui complique leur utilisation en environnement industriel où ils sont, de surcroît, soumis à des contraintes de temps. Dès lors, la conception et l'implantation d'algorithmes parallèles faisant intervenir ces modèles devient incontournable.

Dans ce contexte, les machines parallèles permettent à la fois de traiter des problèmes de taille réelle (size-up) et de les accélerer pour les traiter dans un temps limité (speed-up). Mais en contrepartie, la capacité d'un système parallèle à tolérer les pannes et à réaliser quand même les traitements dans le temps imparti, devient primoridiale.

2.2 Distribution de l'algorithme

2.2.1 Algorithme séquentiel et difficultés de distribution

La figure 2.1 présente l'algorithme séquentiel qui est utilisé pour effectuer la valorisation. Cet algorithme consiste à balayer toute la période de valorisation de la date finale à la date initiale et à déterminer, à chaque étape de ce parcours, l'action la plus intéressante à entreprendre : injecter, sous-tirer ou ne rien faire. La décision prise dépend d'une part des niveaux de stocks dont les valeurs sont conditionnées par différentes contraintes (cf. Section 2.1), d'autre part de la valeur des prix renseignée par le modèle de prix utilisé et finalement des décisions et résultats du pas de temps précédent qui sont sauvegardés dans la structure de données *OldRes* à la fin de chaque itération en temps. Les résultats du pas de temps courant sont stockés dans *NewRes*.

La dépendance qui existe entre les résultats du pas de temps courant et les résultats du pas de temps précédent rendent la parallélisation au niveau de la boucle temporelle impossible. En revanche, une parallélisation de la boucle sur les niveaux de stocks est possible, mais nécessitera un échange complexe de données entre les processus à chaque pas de temps (cf. Section 2.2.2).

2.2.2 Algorithme distribué avec boucles et plan de routage

Stratégie de distribution Afin de réaliser des accélérations importantes et permettre le passage à l'échelle, l'algorithme de la figure 2.1 a été parallélisé pour des architectures extensibles tels les grappes de PCs et les supercalculateurs à mémoire partagée.

```
1 For t := (N − 1)Δt to 0
2     For c ∈ niveaux de stock admissibles
3         For s ∈ valeurs de prix possibles
4             NewRes[s, c] = calcul(OldRes, s, c)
5     //Recopie pour le pas de temps suivant
6     OldRes[∗, ∗] := NewRes[∗, ∗]
```

FIGURE 2.1 – Algorithme séquentiel simplifié de contrôle stochastique.

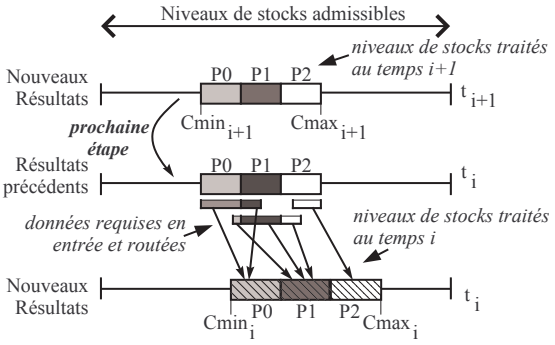


FIGURE 2.2 – Exemple sur trois processeurs d'une redistribution de données optimisée.

Comme le montre la figure 2.2, à l'étape $i + 1$ de la boucle en pas de temps (cf. ligne 1, figure 2.1), chaque processus effectue les calculs pour le sous-ensemble contigu des niveaux de stocks — compris entre $Cmin_{i+1}$ à $Cmax_{i+1}$ — qui lui ont été attribués. À l'étape i (étape suivante), la nouvelle boucle sur les nouveaux niveaux de stock (de $Cmin_i$ à $Cmax_i$) est partagée entre les processus et la répartition des niveaux de stocks a changé. Pour effectuer sa nouvelle tâche, chaque processus a besoin de certains résultats de l'itération précédente dont il n'est pas nécessairement en possession. Ainsi, chaque processus détermine d'une part les données en sa possession dont auront besoin les autres processus, et d'autre part les données qui lui manquent et qui sont détenues par d'autres processus ; ces informations sont alors utilisées pour dresser un *plan de routage*. dans lequel figurent toutes les communications qu'il devra effectuer. Ce plan est finalement exécuté selon un schéma de communication préétabli.

Détails de l'algorithme distribué En suivant la stratégie décrite dans le paragraphe précédent, nous avons conçu l'algorithme distribué décrit dans la figure 2.3. Cet algorithme commence par fixer les valeurs futures du stock de gaz et de son prix à t_n, selon différents scénarios préétablis. Puis il exécute une boucle backward de t_{n-1} à t_0 qui est constituée de deux sous-étapes La première consiste à *planifier et à exécuter l'échange des données*. La seconde correspond à la *nouvelle phase de calculs*.

Pendant la première sous-étape chaque processus commence par déterminer la totalité du nouveau plan de partage des calculs et la nouvelle distribution des données. Les calculs impliqués par cette étape sont très simples (calculs d'indices et de bornes d'intervalles) et ne présentent donc aucun intérêt à être distribués : ils sont donc répliqués et réalisés par tous les processus. Ensuite, chaque processus établit son plan de routage et redimensionne ses tables locales conformément aux besoins de la nouvelle distribution des données. Dans notre cas, chaque processus est capable de déterminer les données dont il a besoin et qu'il doit recevoir des autres, mais aussi celles qu'il possède et qui sont nécessaires aux autres processus, donc celles qu'il doit leur envoyer. Chaque processus détermine ainsi seul la totalité de son plan de routage. Cependant, pour d'autres variantes de l'algorithme, un processus ne pourrait pas déterminer les données requises par les autres, et devrait se faire communiquer ces besoins. L'établissement du plan de routage comporterait alors des sous-étapes de communications courtes et régulières entre les processus. Dans tous les cas, l'échange effectif des données s'effectue immédiatement après et repose sur des communications point-à-point.

La seconde sous-étape consiste en un calcul local à chaque processus suivant l'algorithme de contrôle stochastique partiellement décrit dans la figure 2.1.

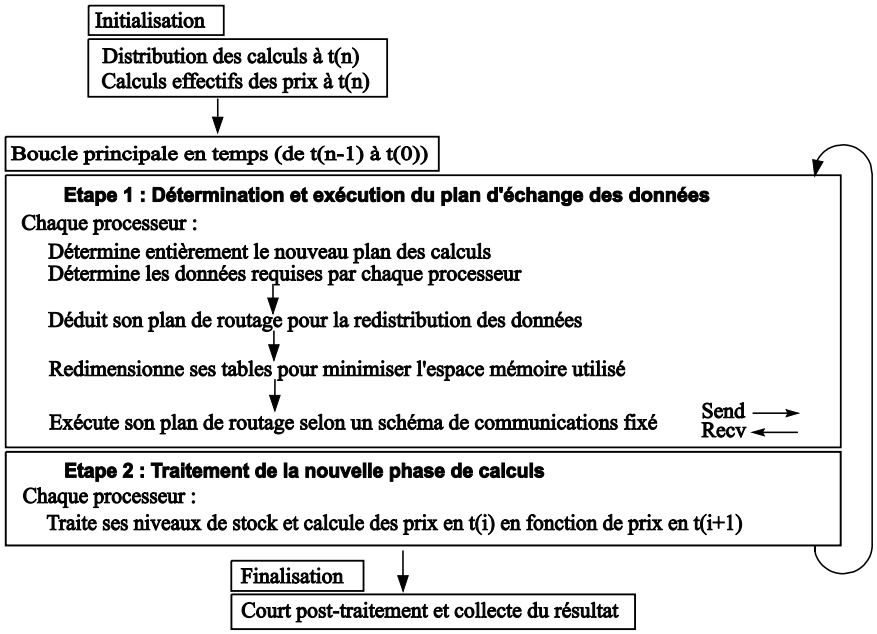


FIGURE 2.3 – Étapes principales de notre algorithme de contrôle stochastique distribué.

Spécificité de l'algorithme L'algorithme parallèle proposé se fonde sur une parallélisation selon le modèle SPMD et admet une structure classique marquée par une boucle de calcul et une alternance d'une phase de calculs avec une phase de communication. La spécificité de l'algorithme est dans l'approche utilisée pour affronter une évolution permanente des données et des calculs à traiter, tout en ne stockant et en ne communiquant que le minimum de données.

Cette approche s'appuie sur des fonctions de partitionnement qui renseignent sur les données possédées et sur les données dont a besoin chaque processus à chaque itération. Ces fonctions permettent d'une part de réaliser d'éventuels redimensionnements. D'autre part, elles permettent de déterminer de manière automatique les communications qui doivent avoir lieu entre les processus.

Il en résulte le concept de plan de routage qui détermine les communications à accomplir et qui laisse le choix sur la manière de les réaliser (schéma de communication). Un exemple de schéma de communication est celui où toutes les communications sont simplement plannifiées pour être exécutées en même temps. Mais dans cette application, nous utilisons un schéma qui « étale » les communications de manière à ne pas saturer le réseau de communications (cf. Section 5.6.2, p. 91).

2.2.3 Implantations

L'algorithme présenté dans la section précédente a donné lieu à deux implantations. La première privilégiait la facilité d'utilisation en combinant Python avec du MPI et du C++. L'avantage de Python est de permettre de facilement paramétrer l'application et d'ajouter rapidement des modules tels que des modules de visualisation.

Quant à la deuxième implantation, elle vise la performance pure et a été écrite entièrement en C/C++ et MPI. Cette implantation offre le choix entre trois couples de routines de communication point-à-point de MPI : `MPI_Bsend/MPI_Recv`, `MPI_Ibsend/MPI_Irecv` et `MPI_Issend/MPI_Irecv`. `MPI_Bsend` et `MPI_Recv` correspondent à des routines de communication dites « bloquantes ». La particularité de `MPI_Bsend` est de laisser le soin à l'utilisateur d'allouer explicitement une mé-

moire tampon (ou *buffer*) suffisamment grande pour pouvoir contenir l'ensemble des données à envoyer. `MPI_Ibsend` et `MPI_Irecv` correspondent aux versions dites « non bloquantes » des routines précédentes. Ces dernières permettent d'effectuer du recouvrement entre les calculs et les communications mais également entre plusieurs communications. La différence de `MPI_Issend` par rapport à `MPI_Ibsend` est la présence d'une synchronisation (ou « *rendezvous* ») entre l'émetteur et le récepteur : les deux doivent être prêts à réaliser la communication.

D'autres routines de communication MPI existent, mais elles ont été écartées par manque de souplesse d'utilisation ou par manque de portabilité. Dans notre cas, en échange d'un léger surcoût de temps de synchronisation, le couple `MPI_Issend/MPI_Irecv` ne nécessite pas d'allocation de buffers de communications supplémentaires et s'est avéré très portable. Cette économie rend la routine `MPI_Issend` très intéressante pour la distribution d'applications gourmandes en mémoire.

2.3 Expérimentation et apparition des pannes

2.3.1 Des benchmarks avec des besoins variés

Selon le modèle de prix évalué, les quantités de calculs à effectuer et de mémoire à utiliser varient beaucoup. Les besoins en CPU et en RAM seront donc très différents d'un benchmark à l'autre. Les modèles de prix utilisés comprenaient :

- un modèle de prix « gaussien à un facteur » : c'est un modèle de référence qui est couramment utilisé de part sa rapidité d'exécution ;
- un modèle de prix « normal inverse gaussien » qui est beaucoup plus lourd que le modèle gaussien en termes de calculs : ceci rend son utilisation moins fréquente ;
- un modèle de prix « gaussien à deux facteurs » : des trois modèles utilisés c'est de loin le modèle le plus lourd aussi bien en termes de calculs qu'en termes d'espace mémoire nécessaire à son exécution.

Plus d'information sur les détails mathématiques de ces modèles est disponible dans [28].

Avec la configuration des modèles prise pour les expériences, les zones de recouvrement sont constantes au sein d'un modèle ce qui conduit à un volume de données échangées constant par processeur et par itération. Le volume de données échangées par itération par l'ensemble des processus augmente linéairement par rapport au nombre de processeurs et nous avons en moyenne sur 32 processeurs 1 Mo de données échangées pour les modèles « gaussien » et « normal inverse gaussien » contre 349 Mo pour le modèle « gaussien à deux facteurs ».

2.3.2 Des plates-formes d'expérimentation variées

En plus de modèles de prix variés, l'évaluation des performances de notre algorithme parallèle a été réalisée sur trois architectures distribuées différentes. Plus particulièrement, l'environnement d'expérimentation comprenait deux grappes de PCs et un supercalculateur :

- la *grappe de Pentium 4* de SUPELEC qui était composée de 32 PCs reliés par un réseau Gigabit-Ethernet bon marché : chaque PC comportait un processeur Pentium 4 à 3 GHz et 2 Go de mémoire ;
- la *grappe de bi-Opteron* de l'INRIA qui disposait de 72 PCs reliés par un réseau Gigabit-Ethernet de très bonne qualité : chaque PC comportait deux processeurs Opteron monocœur à 2 GHz et 2 Go de mémoire ; cette grappe était située sur le site de Sophia de la plate-forme française d'expérimentation Grid'5000 [59] ;

– le *supercalculateur Blue Gene/L* d'EDF R&D doté de 4096 nœuds et d'un réseau d'interconnexion très spécialisé. Ce dernier est constitué d'un réseau principal en tore 3D ainsi que de réseaux auxiliaires pour les communications globales, les entrées/sorties et la gestion de la machine [35]. Sa configuration par nœud est, de loin, la moins puissante parmi les trois architectures considerées – deux processeurs PowerPC à 700 MHz et 1 Go de RAM – mais l'architecture Blue Gene permet d'assembler un grand nombre de ces processeurs : deux par noeud soit 8192 dans notre cas.

2.3.3 Observation des pannes rencontrées

Les modèles de prix et les architectures décrits précédemment ont conduits à de nombreuses expériences au cours desquelles des pannes se sont produites. Les pannes ont été observées surtout sur nos grappes de PCs, et bien que l'origine des pannes n'ait pu être déterminée, elles se sont traduites par l'arrêt de l'application. En l'absence de mécanismes de tolérance aux pannes, l'application doit être relancée depuis le début, et il devient alors impossible de respecter des contraintes de temps sur nos grappes de PCs.

Contrairement aux grappes de PCs utilisées, aucune panne n'est survenue au cours des expériences menées sur le supercalculateur Blue Gene/L. Ce dernier dispose de mécanismes qui permettent entre autres de fournir à l'utilisateur des nœuds très stables et moins sujets à des pannes. Nous pouvons mentionner l'existence de nœuds dédiés uniquement aux calculs et de nœuds dédiés aux E/S. Cette répartition des rôles permet de simplifier la pile logicielle sur les nœuds de calculs qui n'en deviennent que plus fiables. Le nombre moins important de nœuds d'E/S (1 nœud d'E/S pour 8 nœuds de calculs) contribue également à améliorer la fiabilité de l'ensemble du supercalculateur (cf. Section 3.2.2, p. 27). Outre ces choix architecturaux, le supercalculateur Blue Gene/L fournit une bibliothèque de réalisation de points de reprise (sauvegardes) de niveau système et où les sauvegardes sont initiées par l'application (cf. Section 3.7, p. 44).

2.3.4 Résumé des résultats obtenus

Les résultats et performances obtenues avec cette application de valorisation d'un actif de stockage de gaz sont hors du sujet de ce mémoire de thèse. On retiendra seulement que :

– Nous avons obtenu de très bonnes accélérations sur BG/L et sur grappes de PCs avec les plus gros benchmarks utilisant des modèles de prix complexes (modèle « normal inverse gaussien »). Cependant les temps d'exécution étaient encore de plus de 2000s sur 1024 processeurs d'un BG/L ou sur plusieurs centaines de processeurs d'une grappe de PCs, ce qui laisse le temps à une panne de survenir en cas d'exploitation intensive.
– Nous avons obtenu des speedups plus modestes et moins réguliers avec des benchmarks utilisant des modèles de prix plus simples (modèle « gaussien »), notamment sur une grappe de PCs bon marché équipée d'un réseau d'interconnexion Gigabit Ethernet de qualité moyenne.

Finalement, cette parallélisation a permis d'étudier le comportement d'un modèle « gaussien à 2 facteurs » sur des tailles de problèmes qui n'avaient pas pu être traitées auparavant. Pour plus de détails sur les résultats de cette étude, voir [28, 128, 85] ainsi que l'annexe A.

2.4 Résumé de l'étude de cas

Dans ce chapitre nous avons présenté une version parallèle d'un algorithme de contrôle stochastique qui est utilisé pour la valorisation d'actifs de stockage de gaz. L'algorithme parallèle proposé se fonde sur une parallélisation selon le modèle SPMD avec une évolution permanente des quantités de données des calculs à traiter. Par ailleurs, il admet une structure classique marquée par une boucle incluant l'alternance d'une phase de calculs avec une phase de communication, que l'on retrouve dans d'autres algorithmes parallèles. Mais, contrairement à ces derniers il accorde une importance particulière à la minimisation de l'empreinte mémoire et des volumes de communication afin de permettre le passage à l'échelle.

Les expérimentations menées avec nos implantations ont démontré l'efficacité des choix et des techniques de distribution sur des grappes de PCs (jusqu'à 128 nœuds) mais aussi sur un supercalculateur Blue Gene/L d'IBM (jusqu'à 1024 nœuds). Du point de vue des performances uniquement, l'utilisation de grappes de PCs apparaît comme une alternative intéressante à celle de supercalculateurs. Cependant, ces mêmes expérimentations nous ont également confronté à des pannes qui nous ont contraint à relancer les applications depuis le début. Durant nos benchmarks, l'occurrence des pannes était limitée aux seules grappes de PCs, ce qui confirme la meilleure fiabilité du supercalculateur Blue Gene/L.

Le travail mené sur cet d'algorithme financier nous a sensibilisé au problème des pannes. Il a par ailleurs motivé les travaux sur la tolérance aux pannes qui ont suivi. L'approche d'apporter de la tolérance aux pannes au niveau applicatif (cf. chapitre 4) a bénéficié de l'expérience acquise au cours de ce travail. Notamment, pour le cas SPMD, nous retrouvons les concepts de fonctions de partitionnement et de plan de routage dans notre framework FT-GReLoSSS (cf. chapitre 5).

Enfin, notons que cette recherche en collaboration avec EDF s'est poursuivie par le développement d'un algorithme de contrôle stochastique en N dimensions (N stocks d'énergie à exploiter conjointement et au mieux pour répondre à la demande au meilleur prix) [158, 157]. Les solutions algorithmiques adoptées sont une extension à N dimensions de la solution à 1 dimension du problème de valorisation d'un actif de stockage de gaz, et sont aujourd'hui utilisées par EDF et ses filiales dans plusieurs applications. En particulier, nous pouvons citer l'intégration de ces solutions par EDF dans deux outils d'étude et un outil de production par sa filiale anglaise *EDF Energy*. Dans l'ordre, le premier outil d'étude concerne l'allocation optimale de fonds pour le démantèlement de centrales nucléaires, tandis que le second concerne la gestion d'un parc de production d'électricité sous contrainte de satisfaction de la demande. Enfin, l'outil de production est utilisé pour la valorisation d'actifs thermiques.

Les temps d'exécutions sur 32000 cœurs d'un Blue Gene/P ou sur quelques milliers de cœurs de grappes de Intel Xeon Nehalem restent encore importants, et la problématique de la tolérance aux pannes émerge progressivement. Les mécanismes de tolérance aux pannes développés dans cette thèse pourraient aussi s'appliquer à ces algorithmes distribués N-dimensionnels.

Chapitre 3

État de l'art en tolérance aux pannes dans les grappes de PCs

Un *système distribué* [70, 109] peut être défini physiquement comme un ensemble d'unités de calcul autonomes (ou nœuds) qui sont reliées entre elles par un réseau de communication. Cet ensemble physique est exploité par un ensemble logique de processus qui communiquent entre eux uniquement par passage de messages. Une application distribuée est constituée de processus qui s'exécutent sur les nœuds du système distribué et qui communiquent afin d'atteindre un but commun.

Cette définition très générale peut englober de nombreux systèmes. Nous pouvons citer Internet, les réseaux locaux, les réseaux de capteurs, etc. Nous nous intéressons plus particulièrement aux systèmes distribués qui sont spécialisés en calcul. Un exemple de tel système est Intercell, la grappe de 256 nœuds hebergée sur le campus de Supélec à Metz (cf. Section 7.1.5, p. 118). Ces systèmes distribués sont exploités dans le but de permettre l'exécution d'applications qui ne pourraient pas être éxecutées sur un seul nœud. Ces exécutions sur un seul nœud (ou séquentielles) peuvent être impossibles pour des raisons de temps d'exécution ou d'insuffisance mémoire. Dans le premier cas on cherche à obtenir du *speed-up* (ou accéleration) tandis que dans le second cas on cherche à obtenir du *size-up* (ou passage à l'échelle). Dans certains cas on visera les deux objectifs. Le processus qui transforme l'application séquentielle en application distribuée (donc adaptée à s'exécuter sur des systèmes distribués) se nomme communément distribution ou parallélisation.

Ces systèmes sont complexes et, de part la nature imparfaite des éléments qui les constituent, sont sujets à des pannes. L'augmentation du nombre de composants dans ces systèmes a une influence directe sur l'occurrence des pannes qui deviennent alors très courantes. Cette situation appelle donc à la mise en place de solutions adaptées de tolérance aux pannes.

3.1 Concepts de tolérance aux pannes

3.1.1 Définitions : pannes, erreurs et fautes

La ***tolérance aux pannes*** est un domaine très vaste qui traite des pannes qui surviennent dans un système. Pour un système distribué c'est la capacité à fonctionner en présence de pannes de ses composants [109]. Rendre un système tolérant aux pannes revient à le rendre tolérant aux pannes de ses composants. En particulier, un système ne peut être rendu tolérant à sa propre panne [109].

Une ***panne*** désigne l'incapacité d'un élément à accomplir sa tâche en raison d'erreurs qui surviennent au sein de cet élément ou dans son environnement. Une panne est souvent définie comme un écart visible du comportement d'un système par rapport à sa spécification [98, 7, 45]. Les pannes résultent de problèmes inattendus et internes à un système qui finissent par se manifester dans le comportement externe du système. Ces problèmes sont appelés des ***erreurs***. À l'origine de ces erreurs sont des ***fautes*** qui peuvent être de natures très variées.

Les systèmes sont composés de plusieurs composants qui sont eux-mêmes des systèmes. À cause de cette imbrication de systèmes, il en résulte la relation générale entre pannes, fautes et erreurs qui est illustrée à la figure 3.1. Il est important de noter que la panne peut se manifester sur un autre composant (du système) que le composant fautif. Cette manifestation permet en général de détecter la panne mais n'est pas toujours suffisante pour en localiser l'origine exacte (*i.e.* : la faute et le composant fautif).

L'exemple de la corruption mémoire proposé dans [75] illustre bien la relation complexe qui peut exister entre faute, erreur et panne. La corruption d'un emplacement mémoire par une particule alpha est une faute. Si cet emplacement mémoire contient une donnée, alors cette donnée corrompue constitue une erreur. Un arrêt inattendu du programme (ou la production d'un résultat erroné) suite à la corruption de cette donnée est une panne. Ainsi, une faute ne résulte pas nécessairement en une erreur et une erreur n'aboutit pas nécessairement à une panne. Enfin, les systèmes actuels sont tellement complexes qu'il peut être hasardeux de se prononcer sur la cause exacte d'une panne.

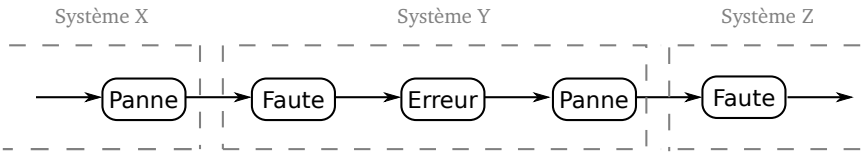


FIGURE 3.1 – Relation entre erreur, faute et panne.

3.1.2 Classifications de fautes/pannes

Les fautes peuvent être classées de plusieurs manières. En 2004 [8] propose une classification des fautes qui peuvent affecter un système pendant sa vie. Cette classification comporte huit classes élémentaires qui permettent de caractériser les fautes de manière assez précise. Les fautes peuvent être classées en fonction de la phase pendant laquelle elles surviennent : ce peut être pendant la phase de développement ou pendant la phase d'utilisation du système. Les fautes peuvent être extérieures ou intérieures au système, etc. Comme les pannes sont finalement la manifestation de fautes, les pannes peuvent suivre une classification directement en fonction de la faute qui l'a provoquée. Nous formulons ci-après deux telles classifications de pannes qui sont dérivées des classifications de fautes.

La première classification prend en compte la durée (ou persistence temporelle) des pannes [8, 31, 130] et donne lieu à des ***pannes permanentes*** ou des ***pannes temporaires***. Une panne permanente est une panne qui continue d'exister jusqu'à ce que la faute qu'il l'a provoquée soit corrigée. Au niveau logiciel il s'agit typiquement d'un bogue dans un programme : cette faute est reproductible et continuera d'exister tant qu'elle ne sera pas corrigée par le programmeur. Au niveau matériel, une panne permanente correspond généralement à un changement irréversible d'un composant physique. La mort d'un disque dur est un exemple de panne permanente : il devra être remplacé. À l'inverse, une panne transitoire ne subsiste pas et disparaît. Souvent la cause de la panne n'est pas clairement identifiée mais disparaît en réessayant. Un nœud peut s'arrêter

de fonctionner correctement sans raison apparente et être rétabli par simple redémarrage. De la même manière, un logiciel peut s'arrêter de fonctionner correctement à cause de l'occurrence rare de plusieurs évènements. Une simple réinitialisation suffit alors à faire disparaître la panne.

La seconde classification de pannes s'effectue selon le niveau d'occurrence de la faute : matériel ou logiciel. Cette classification distingue donc les ***pannes matérielles*** des ***pannes logicielles***. Nous parlerons de panne matérielle (resp. logicielle) lorsque le composant fautif est matériel (resp. logiciel) : l'origine (ou cause) de la panne est alors matérielle (resp. logicielle). Cette distinction apparaît clairement dans les exemples donnés pour la première classification.

Dans les systèmes distribués, une classification répandue est celle qui se fonde sur le comportement du composant en cas de panne [39, 40, 109] et qui apparaît à partir de 1985. La panne peut être :

- une *panne par arrêt* (*crash failure*) : le composant s'arrête aussitôt ;
- une *panne par omission* : le composant omet de répondre à certaines entrées [1] ;
- une *panne de timing* (ou de performance) : le composant ne répond pas dans les temps (soit trop tôt, soit trop tard) ; ou
- une *panne byzantine* : le composant se comporte de manière totalement arbitraire pendant une panne.

Cette classification est completée en 1994 dans [109] avec l'ajout des *pannes de correction* : le composant produit un résultat incorrect en réponse à des entrées particulières ou suite à des erreurs survenues pendant le traitement. Ce type de pannes fait partie des ***pannes silencieuses***. Le terme a été introduit en 2005 dans [75] pour caractériser plus généralement le comportement de composants qui tombent en panne mais ne le transmettent pas aux autres composants.

Cette dernière classification est notamment la fondation des modèles de fautes que nous présentons à la section 3.1.3 et qui sont entre autre utilisés pour faciliter la présentation de différentes techniques de tolérance aux pannes.

3.1.3 Modèles de fautes

Dans la littérature plusieurs modèles de fautes ont été proposés. Chacun de ces modèles décrit le(s) comportement(s) adopté(s) par un composant lorsqu'il tombe en panne. Les deux modèles les plus répandus sont :

- le ***modèle de fautes par arrêt*** (*fail-stop fault model*) introduit à partir de 1983 [131, 133, 134, 132] suppose qu'en réponse à une faute, le composant fautif se met dans un état qui permette aux autres composants de détecter l'occurrence de la panne puis s'arrête ;
- le ***modèle de fautes byzantin*** introduit à partir de 1982 [80] ne fait aucune supposition quant au comportement d'un composant fautif : ce dernier peut se comporter de manière arbitraire. En particulier, il peut communiquer des informations conflictuelles à différents composants d'un système.

Chacun de ces modèles décrit une classe de pannes et facilite l'exposition de techniques de tolérance aux pannes. L'absence d'*a priori* sur le comportement des composants fautifs rend le modèle byzantin adapté dans certains contextes tels que la sécurité. Le traitement de telles pannes repose sur le résultat établit en 1982 par Lamport, Shotak et Pease [80], selon lequel aucune garantie sur le fonctionnement correct d'un système ne peut être apportée si au moins 1/3 des composants présente un comportement byzantin. Tolérer des fautes byzantines nécessite donc une forte réplication des composants et une telle généralité est souvent exagerée et inadéquate dans des systèmes où l'occurrence de telles pannes est faible. Notamment la généralité du modèle

1. Une panne par arrêt est un cas particulier de panne par omission dans lequel le composant fautif omet de répondre à toutes les entrées.

byzantin rend difficile son utilisation dans des environnements où une grande majorité des pannes se manifestent par un arrêt. Dans ces environnements, l'utilisation du modèle par arrêt est plus adaptée et donc préferée.

Bien que de nombreuses pannes rentrent dans le modèle par arrêt, ce dernier ne permet pas de capturer le comportement de pannes un peu plus complexes telles que les pannes de performance ou les pannes silencieuses. Le modèle a donc été jugé comme trop simpliste par rapport à la réalité. Pour y remédier, deux autres modèles ont été proposés en complément :

- le ***modèle de fautes par bégaiement*** (*fail-stutter fault model*) [6] établi en 2001 se fonde sur l'observation que des composants matériels ou logiciels peuvent avoir des performances très faibles tout en continuant à fonctionner correctement. Cet éloignement des performances par rapport aux spécifications est suffisant pour être consideré comme une panne. Le modèle de fautes par bégaiement peut être intéressant dans une optique préventive. En effet, une baisse de performance permanente peut être le signal d'une panne imminente. Le composant fautif une fois detecté peut alors être remplacé.
- le ***modèle silencieux de fautes par bégaiement*** (*silent fail-stutter fault model*) [75] établit en 2005 se fonde sur l'observation que de nombreux composants peuvent tomber en panne de manière silencieuse et que ce type de comportement n'est pris en compte ni par le modèle de fautes par arrêt ni par celui par bégaiement. La production d'un résultat erroné ou l'absence d'avancement d'une application suite à une faute silencieuse sont deux exemples de pannes silencieuses pour lesquelles les auteurs du modèle proposent une méthodologie de détection [75].

Les auteurs de ces modèles dénoncent la faiblesse des systèmes actuels parce qu'ils sont contruits majoritairement autour du modèle de fautes par arrêt. De ce fait, ils sont pris en défaut lorsque des composants exhibent des pannes de performance ou silencieuses. Malgré le réalisme accru qu'offrent ces modèles par rapport au modèle de faute par arrêt, ces modèles requerraient une mise en œuvre plus complexe. Par exemple, pour détecter des pannes de performance il faudrait arriver à mettre au point des spécifications de performance des composants d'un système. Nous pensons que ceci est loin d'être évident dans la mesure où des performances imprévisibles peuvent survenir assez fréquemment et que leur occurrence peut varier d'un environnement à un autre. Le concepteur de systèmes qui prennent en compte les pannes de performance devrait donc avoir un bon modèle qui puisse capturer la fréquence d'occurrence et la durée de ces pannes de performance. De surcroît, il faudrait mettre en place un mécanisme de surveillance pour pouvoir contrôler les écarts et par la suite décider de la bonne action à suivre.

Finalement, la difficulté de prendre en compte des pannes de performance et silencieuses d'une part, et la prépondérance des pannes par arrêt dans la plupart des systèmes d'autre part, expliqueraient en partie le manque d'acceptation des deux modèles par bégaiement.

3.2 Les pannes et leur tolérance dans les grappes de PCs

3.2.1 Architecture et utilisation des grappes de PCs

Les grappes de machines auxquelles nous nous intéressons comportent des nœuds composé de matériel standard (grand public) et reliés par un réseau privé rapide. Ces grappes peuvent s'apparenter à des grappes de type *Beowulf* [143, 12] dans la mesure où l'architecture est similaire et l'objectif recherché est la performance ; cette dernière est comparable à la performance delivrée par certains supercalculateurs mais pour un coût beaucoup moins important. L'architecture de telles grappes comprend en général quelques nœuds frontaux dédiés à des tâches de gestion ainsi que les nœuds de calcul dont le nombre peut s'élever à plusieurs centaines voire milliers [68]. Les

nœuds de calcul sont souvent homogènes et comportent au moins une unité de calcul (processeur ou cœur), de la mémoire vive, une ou plusieurs interfaces réseaux et un disque dur. Les tâches de gestion sur les nœuds frontaux comprennent en général un mécanisme d'identification, un système de réservation de ressources et éventuellement un système de plannification. Ces nœuds peuvent également être reliés à un système de stockage spécialisé.

3.2.2 Origine et exemples de pannes

Pour satisfaire des besoins grandissants en puissance de calcul, les grappes de calcul connaissent une croissance importante du nombre de leur nœuds. Les applications des utilisateurs peuvent alors bénéficier de plus de nœuds pour s'exécuter plus rapidement mais également pour traiter des problèmes plus gros. La principale conséquence de la multiplication du nombre de nœuds est une baisse de la fiabilité de la grappe. Cette dernière devient alors davantage sujette à des pannes de ses composants (nœuds et réseau d'interconnexion). En effet, l'étude [136] menée sur les pannes survenues dans les grappes de calcul au LANL (Los Alamos National Laboratory) conclut que les taux de pannes sont approximativement proportionnels au nombre de nœuds du système ; ceci indique que les taux de pannes suivent une croissance linéaire par rapport à la taille du système. Par ailleurs, le fait que les taux de pannes dans une grappe dépendent principalement de sa taille (nombre de nœuds) et beaucoup moins du type de matériel utilisé [136] soutient le fait que la croissance de la taille des grappes de PCs est significativement plus importante que l'amélioration individuelle du niveau de fiabilité de chacun de ses nœuds [137].

Un autre facteur qui contribue à l'occurrence des pannes provient de la complexité des matériels et des logiciels qui font vivre la grappe. Au niveau matériel, les nœuds et leur composants sont agencés de manière très compacte. De leur côté, les composants deviennent de plus en plus denses : par exemple, on intègre de plus en plus de transistors et de cœurs par circuit intégré. Le refroidissement du système devient alors une préoccupation importante. En effet, les surchauffes, liées par exemple à une période d'activité intense, peuvent rendre le système instable et conduire à des pannes (par arrêt). Pour éviter que des composants ne soient endommagés de façon irreversible, on dote le système de mécanismes de protection ou prévention qui détectent ces situations anormales et éteignent ou ralentissent le nœud : ces mécanismes forcent en quelque sorte une panne. Les composants matériels sont également soumis au phénomène naturel de l'usure notamment à cause du stress mécanique et thermal. Finalement, la densité rend ces composants plus vulnérables aux attaques de particules alpha et aux rayons cosmiques [164]. Ces attaques couplées à une réduction des voltages de fonctionnement dans le but de réduire la consommation énergétique favorisent l'occurrence d'erreurs douces (ou *soft errors*) tels que les *bit-flips* dans les mémoires.

À l'image des composants matériels, les composants logiciels connaissent également une complexification. Par exemple, le noyau linux, qui est utilisé dans la majorité des grappes de PCs, avait déjà connu en 2006 une augmentation de sa taille de plus d'un facteur 38 depuis 1994 [77]. De plus, les environnements de grappes font intervenir et intéragir plusieurs autres logiciels dont le plannificateur de tâches, différents intergiciels, le système d'exploitation, le logiciel de supervision, etc. Ces systèmes sont également amenés à s'exécuter pendant de longues périodes ce qui favorise le phénomène de *vieillissement logiciel* : les conditions propices au déclenchement d'erreurs s'accumulent avec le temps et la charge du système [65, 154]. Ce type de pannes peut être prévenu en remettant le système dans un état propre (ou sain). Le redémarrage (ou réinitialisation) d'un système est un exemple de *rajeunissement* du logiciel (ou *software rejuvenation*).

Les supercalculateurs *BlueGene/L* d'IBM, qui peuvent comporter jusqu'à 65536 nœuds de calcul, atteignent un temps moyen entre des pannes (ou Mean Time Between Failures (MTBF))

de 6, 5 jours [110] alors que les scientifiques ne prévoyaient que quelques minutes pour un tel nombre de composants. Certes les composants utilisés possèdent un niveau de fiabilité au-dessus de la moyenne mais le niveau de fiabilité global atteint est attribué à l'application du principe de simplicité [110]. En particulier, les fonctionalités des nœuds de calcul sont réduites au minimum. Le reste des fonctionnalités est repoussé sur les nœuds d'entrées/sorties qui sont au nombre de 1024. L'environnement de programmation suit également cette philosophie. Enfin, même des mesures contre le vieillissement logiciel sont adoptées sur les nœuds de calcul : le système logiciel d'exploitation des nœuds de calculs (ou *Compute Node Kernel*) est redéployé à chaque réservation.

Plusieurs sources [136, 139, 94] s'accordent sur le fait que les éléments qui provoquent des pannes se situent au niveau matériel et logiciel des nœuds. Des pannes réseau existent également mais sont en comparaison faibles [136, 94]. Une partie des pannes est d'origine non identifiée [136]. Plus précisement, au niveau matériel, les coupables sont majoritairement les processeurs, les mémoires et les disques durs [136, 139, 94].

Pour résumer, les pannes dans les grappes de PCs résultent du nombre de nœuds ; de la complexité des logiciels et matériels et de la complexité des interactions qui en résultent. Alors que les origines des pannes peuvent être extrêmement complexes et pas toujours élucidées, pour l'utilisateur, une panne entraîne souvent l'interruption de l'exécution de son application.

3.2.3 Dilemme de réplication : spatiale ou temporelle ?

Pour faire face à la multitude de pannes qui guettent les applications s'exécutant dans les grappes de PCs, plusieurs solutions de tolérance aux pannes ont été proposées dans la littérature. Ces dernières se construisent en général dans le cadre d'un modèles de fautes. Les plus communs sont les modèles de faute par arrêt et le modèle de fautes byzantin exposés à la section 3.1.3. Par ailleurs les solutions de tolérance aux pannes se distinguent par la forme de redondance qu'elles exploitent : il peut s'agir d'une redondance en espace ou en temps. La redondance en espace se fonde sur l'utilisation de ressources supplémentaires - celles-ci sont souvent physiques - tandis que la redondance en temps se fonde sur l'utilisation de temps de calcul supplémentaire.

De part la nature et la fréquence des pannes qui surviennent dans les grappes de PCs (Section 3.2.2) le modèle de fautes par arrêt est plus pertinent que le modèle de fautes byzantin. En ce qui concerne le type de redondance, les deux types de redondance citées précédemment sont possibles. Une manière d'accroître la fiabilité de la grappe consiste à rendre chacun des nœuds qui la constitue plus fiable. Ceci peut être atteint en effectuant la réplication au niveau du nœud entier. Cette réplication pourrait ne pas se limiter uniquement aux composants matériels mais s'étendre au logiciel par de la réplication de processus. En 1994, le projet *Delta-4* [118] proposait une architecture visant à apporter de la tolérance aux pannes en utilisant de la réplication au niveau du nœud entier et des mécanismes de réplication (spatiale) de processus. Une autre approche consiste à limiter la réplication au sein des nœuds aux composants importants et les plus sujets à des pannes. Nous retrouvons ce type d'approche dans certains secteurs d'activités comme l'industrie financière pour l'exécution d'applications critiques. [117] fait mention de la mise en place d'une telle grappe en utilisant pour nœuds 60 serveurs ultra-fiables fondés sur l'architecture *System X* d'IBM [159]. Cette architecture bénéficie d'une redondance très poussée au niveau de plusieurs composants (ventilateur, alimentation, mémoire, disque dur, interface réseau, processeur).

L'inconvénient de ces approches vient du surcoût financier et de la sur-utilisation de ressources qu'elles engendrent. Dans les grappes de PCs visant la performance, ces approches ont du mal à se généraliser à tous les nœuds. Une mobilisation double de ressources est de manière très compréhensible perçue comme un gaspillage. En conséquence, la réplication spatiale est employée

pour améliorer la disponibilité des nœuds frontaux de la grappe et ainsi assurer la continuité de l'accès aux ressources de calcul [45, 81]. Pour les nœuds de calcul, les solutions fondées sur de la réplication en temps sont préferées. En l'occurrence, suite à une panne, l'application (distribuée) est amenée à reprendre son exécution seulement à partir d'un état antérieur à son état au moment de la panne. Ce retour en arrière évite la ré-exécution de l'application depuis son état initial et minimise la quantité de calculs perdus. Une présentation générale de ces techniques dites de *reprise par retour arrière* fait l'objet de la section suivante.

3.2.4 Fondements des techniques de reprise par retour arrière

3.2.4.1 Principes et modèle adopté

Les techniques dites de *reprise par retour arrière* (ou *rollback recovery*) modélisent les applications distribuées comme un ensemble de processus qui communiquent à travers un réseau et ce, uniquement en s'échangeant des messages [43]. L'ensemble des processus a accès à un *support de stockage stable* (ou *stable storage device*) lequel constitue une idéalisation d'un support de stockage (*e.g.* : un disque dur). En effet, il est supposé infaillible. Ce support est utilisé par les processus pour y stocker les informations nécessaires à une éventuelle reprise (ou informations de reprise). En cas de panne, ces informations sont utilisées pour reprendre l'application (et donc les calculs) à partir d'un état intermédiaire. De cette manière on réduit la quantité de travail perdue à cause de la panne. Les informations de reprise comportent au minimum un *point de reprise*[2] (ou PDR) de chaque processus. En fonction de la technique de reprise par retour arrière employée, les informations de reprise peuvent également comporter des enregistrements de messages (ou *logs de messages*).

Les *protocoles de reprise par retour arrière* (ou *rollback-recovery protocols*) sont les entités responsables de la sauvegarde des informations de reprise et de leur exploitation au moment de la reprise sur panne. Le rôle de ces protocoles est d'assurer que l'application soit remise dans un état cohérent après une panne. De manière informelle, une reprise sur panne est correcte si les résultats de la ré-exécution de l'application correspondent à des résultats que l'on peut obtenir pendant une exécution sans pannes [34].

3.2.4.2 État cohérent d'une application distribuée

L'état global d'une application distribuée communiquant par passage de messages est constitué de l'état de chacun de ses processus. La figure 3.2 représente l'exécution au cours du temps d'une telle application. Cette dernière est constituée de deux processus P et Q qui s'échangent un message m. On y représente également par des formes rectangulaires noires des états possibles des processus P et Q à différents moments de leur exécution. $EQ1$ et $EQ2$ représentent des états possibles de Q respectivement avant et après l'envoi du message m. $EP1$ et $EP2$ représentent des états possibles de Q respectivement avant et après la réception du message m.

Un état global est *cohérent* s'il représente un état que peut atteindre l'application pendant une exécution correcte et sans pannes. En particulier, c'est un état dans lequel, pour chaque message échangé entre deux processus, si l'état du processus récepteur reflète la réception du message alors l'état du processus initiateur (de l'envoi) doit en refléter l'envoi [34]. Le non respect de cette contrainte conduit à des états globaux incohérents : ce sont des états dans lesquels un message peut avoir été reçu sans avoir été envoyé.

2. Il s'agit d'une sauvegarde du processus également connue sous le terme anglais de *checkpoint*

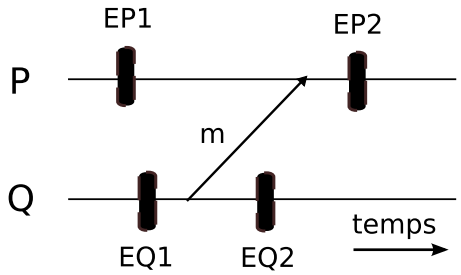


FIGURE 3.2 – Exemple d'exécution de deux processus P et Q s'échangeant un message m. $EP1$, $EP2$, $EQ1$ et $EQ2$ sont des états possibles de ces processus au cours de leur exécution.

L'état global noté $\{EQ1, EP2\}$ et formé par les états $EQ1$ et $EP2$ des processus P et Q, est un état global incohérent : l'état $EP2$ reflète la réception du message m mais l'état $EQ1$ n'en reflète pas l'envoi. On appelle ce type de messages des *messages orphelins* [43] ou encore des *messages en avance*.

L'état global $\{EP1, EQ2\}$ est un état cohérent : il correspond à la situation réelle où le message m a été envoyé par Q mais n'a pas encore été reçu par P. On appelle ce type de messages des *messages en transit* [43] ou encore des *messages en retard*.

Les états globaux $\{EQ2, EP2\}$ et $\{EQ1, EP1\}$ sont cohérents et correspondent à des cas triviaux. Dans l'état global $\{EQ2, EP2\}$, l'envoi et la réception y sont tous les deux reflétés : on parle dans ce cas de *message passé*. Dans l'état global $\{EQ1, EP1\}$, ni l'envoi, ni la réception ne sont reflétés : on parle alors de *message futur*.

Un état global forme une *ligne de reprise* (ou *recovery line*), s'il est cohérent ou, de manière plus générale, s'il est possible à partir de cet état et éventuellement d'informations supplémentaires sauvées au cours de l'exécution normale de reconstituer un état global cohérent.

3.3 Classification des techniques de reprise par retour arrière

Parmi les techniques de reprise par retour arrière, il existe d'une part celles qui se contentent simplement de réaliser des sauvegardes de l'application distribuée et d'autre part, il existe celles qui enregistrent entre deux sauvegardes consécutives de l'application distribuée les messages échangés par les processus. Les protocoles utilisés par ces deux familles s'appellent respectivement *protocoles de points de reprise* (*checkpointing protocols*) et *protocoles par enregistrement de message* (*message logging protocols*).

Une différence notable s'observe en cas de panne. En effet, avec les protocoles de points de reprise, tous les processus doivent effectuer un retour arrière. Ce n'est pas nécessairement le cas des protocoles par enregistrement de message. Par exemple, dans la figure 3.3 si les trois processus P, Q et R utilisent un protocole de point de reprise, ils devront tous reprendre leur exécution à partir de la ligne de reprise formée par les points de reprise CP, CQ et CR. En revanche, s'ils utilisent un protocole d'enregistrement de messages, un retour arrière de tous les processus peut être évité et l'application peut être reprise dans un état plus récent que celui formé par les derniers points de reprise de chaque processus. En supposant que le protocole par enregistrement de messages ait enregistré l'ordre de réception et le contenu des messages, il suffit que le processus Q qui tombe en panne reprenne son exécution à partir de son point de reprise CQ. Pendant la ré-exécution de Q le protocole veillera d'une part à ce que les messages $m6$ et $m7$ soient rejoués pour Q et d'autre part, il veillera à ce que le message orphelin $m5$ ne parvienne pas une seconde fois à P.

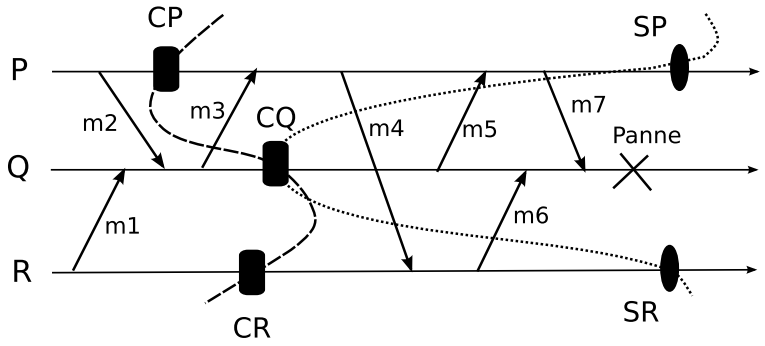


FIGURE 3.3 – Ligne de reprise d'un protocole de points de reprise $\{CP, CQ, CR\}$ et ligne de reprise d'un protocole par enregistrement de messages $\{SP, CQ, SR\}$.

La figure 3.4 illustre la classification des protocoles de points de reprise par retour arrière que nous proposons. Elle est issue de classifications antérieures [43, 22], mais est plus approfondie au niveau des protocoles de points de reprise coordonnés, où se situent nos travaux (cf. Chapitre 4).

Au sein de la famille des protocoles de points de reprise (partie droite de la figure 3.4), on distingue les protocoles de points de reprise non-coordonnés de ceux qui induisent une forme de coordination. Les protocoles sans coordination laissent les processus réaliser des points de reprise de manière indépendante : aucune action n'est entreprise en l'absence de pannes pour obtenir des points de reprise qui forment un état global cohérent. La détermination d'un tel état s'effectue lors de la reprise sur panne. À l'inverse, les protocoles de point de reprise coordonnés utilisent un mécanisme pour coordonner l'action des processus au moment de la sauvegarde. Il en résulte un ensemble de points de reprise qui constitue un état global cohérent, et une reprise sur panne qui s'effectue plus simplement.

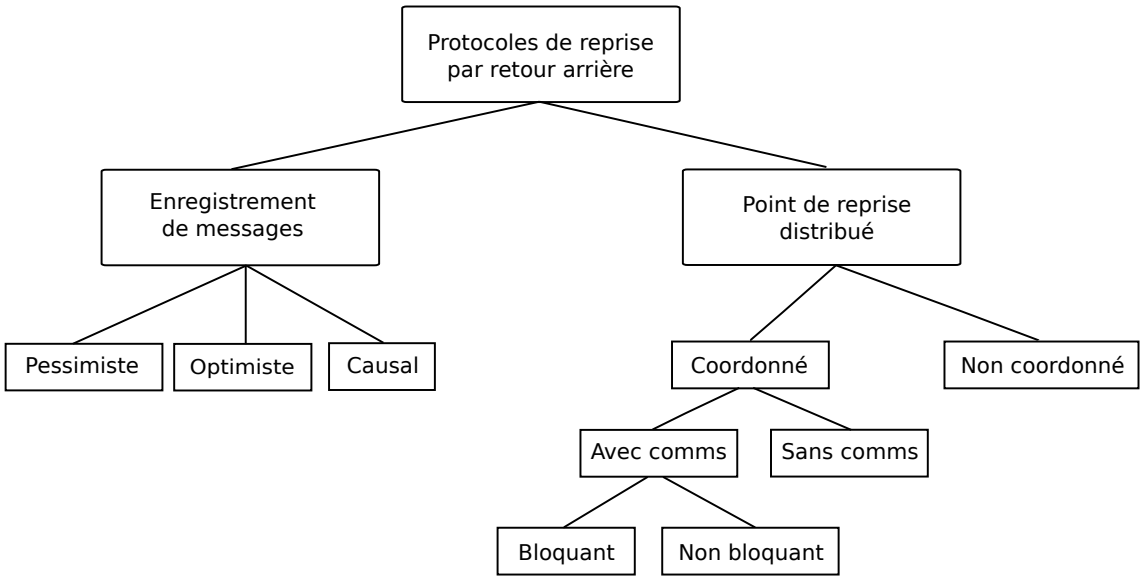


FIGURE 3.4 – Taxonomie des protocoles de reprise par retour arrière.

Par rapport aux classification antérieures, nous introduisons une différentiation des protocoles coordonnés, selon le type de coordination qu'ils utilisent. La coordination peut être effectuée en introduisant des communications entre les processus ou dans certains cas particuliers sans

communication [3]. Par ailleurs, la coordination peut être bloquante ou non-bloquante. Dans le premier cas, l'exécution de l'application est suspendue jusqu'à la constitution d'un état global cohérent. Dans le second cas, l'application n'est pas interrompue et continue à envoyer et à recevoir des messages pendant que le protocole s'exécute.

La famille des procotoles par enregistrement de messages (partie gauche de la figure 3.4) comporte trois principales variantes selon la stratégie d'enregistrement de messages utilisée par le protocole : pessimiste, optimiste et causale, détaillées plus loin.

La section 3.4 est consacrée à une présentation plus approfondie de différents protocoles de points reprise tandis que les différentes variantes des protocoles par enregistrement de message seront présentées de manière plus détaillée à la section 3.5.

3.4 Protocoles de points de reprise pour applications distribuées

3.4.1 Protocoles non coordonnés

Dans les techniques de points de reprise non coordonnés (ou indépendants), chaque processus réalise un point de reprise au moment où il le souhaite. Ceci offre une certaine flexibilité. Suite à une panne, les processus sont redémarrés à partir de leur sauvegarde la plus récente.

Étant donné qu'aucune coordination n'est effectuée, rien ne peut garantir que l'ensemble des sauvegardes individuelles de chaque processus forme une ligne de reprise. Ceci est d'autant plus vrai pour les applications distribuées qui mettent en jeu de nombreuses communications. Il en résulte le besoin de conserver plusieurs sauvegardes par processus et la mise en place de protocoles pour déterminer la ligne de reprise la plus récente. Afin de minimiser l'espace disque occupé par les sauvegardes, des ramasse-miettes sont mis en place.

Par ailleurs, ces protocoles sont sujet à l'*effet domino* [120] dont le résultat est un retour arrière très éloigné par rapport aux sauvegardes les plus récentes. Dans le pire cas, l'effet domino amène l'application dans son état initial : elle est donc contrainte de reprendre son exécution depuis le début et tout calcul effectué jusqu'au moment de la panne se retouve perdu. La figure 3.5 illustre l'effet domino : lorsqu'une panne interrompt le processus P, les processus P et Q effectuent un retour arrière sur leur sauvegardes les plus récentes $CP2$ et $CQ2$ respectivement. Cependant, $\{CP2, CQ2\}$ ne forme pas un état cohérent étant donné que le message $m4$ a été reçu sans avoir été envoyé. Le processus Q retourne donc à l'état $CQ1$. Ceci résout l'incohérence par rapport au message $m4$ mais $\{CP2, CQ1\}$ n'est pas cohérent par rapport au message $m3$. En continuant cette recherche d'une ligne de reprise, les processus P et Q atteignent leurs états initiaux.

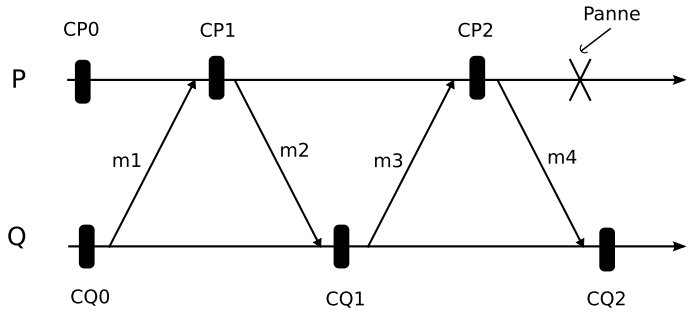


FIGURE 3.5 – Exemple d'effet domino avec deux processus.

3. Les protocoles coordonnés mais sans communication sont souvent de niveau applicatif.

Pour remédier aux effets indésirables résultant des protocoles de points de reprise non coordonnés, des protocoles de points de reprise coordonnés ont été proposés. L'idée derrière ces protocoles est d'utiliser une forme de coordination pour réaliser directement un ensemble de points de reprise qui forme un état global cohérent.

3.4.2 Protocoles à coordination bloquante

Les protocoles de points de reprise coordonnés bloquants font en sorte qu'au moment de la réalisation des points de reprise il n'y ait aucun message de l'application qui circule dans le réseau. Au delà de cette règle, plusieurs types de protocoles à coordination bloquante existent.

3.4.2.1 Le protocole Sync-aNd-Stop (SNS)

La version la plus simple de ce type de protocoles est celle du *Sync-aNd-Stop* [112] de 1993. Dans ce protocole, les processus se synchronisent d'abord pour créer un état global cohérent avant de réaliser les points de reprise. Le protocole est initié par un processus particulier de l'application que l'on appelle coordinateur. Ce dernier diffuse au moment de l'initiation une requête de sauvegarde (checkpoint). Un processus qui reçoit cette requête arrête sont exécution et attend que tous les messages qu'il a envoyés aient été reçus. Ensuite, il notifie le coordinateur qu'il est prêt à réaliser son point de reprise. Lorsque le coordinateur a reçu les notifications de tous les processus, il les informe de procéder à la réalisation de leur point de reprise. Lorsque le coordinateur reçoit à nouveau des notifications de tous les processus, il valide l'ensemble des points de reprise et diffuse une requête pour la reprise l'exécution.

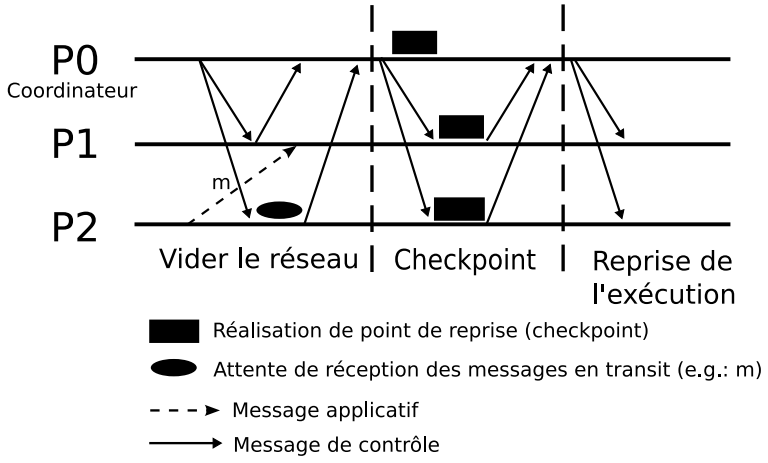


FIGURE 3.6 – Le protocole Sync-aNd-Stop en action.

Le protocole décrit présente quelques inconvénients liés à la synchronisation forte qu'il utilise. En raison d'un nombre très élevé de processus participant, la synchronisation peut générer un très grand nombre de messages. Par ailleurs, si le coût des communications est important, les processus peuvent rester bloqués très longtemps en attendant la disparition des messages en transit dans le réseau. Enfin, la synchronisation entraîne l'écriture quasi-simultanée des points de reprise ce qui, dans le cas de support de stockage commun, peut mettre une pression importante au niveau du système d'entrées/sorties.

Les mesures de performance réalisées dans [24], dans le cadre d'une comparaison entre protocoles bloquants et non-bloquants, confirment que les protocoles bloquants sont principalement

intéressants lorsque l'environnement d'exécution dispose d'un réseau rapide et supporte des montées en charges du système d'entrées/sorties.

Cependant, par rapport à d'autres protocoles, les protocoles de coordination bloquants possèdent l'avantage d'être très simples et faciles à implanter. Ceci expliquerait les nombreuses implantations et variantes qui existent [4, 115, 24, 142, 129, 67].

3.4.2.2 Le protocole de Koo Et Toueg (KET)

Le protocole *KET* proposé en 1987 dans [76] est un protocole de coordination bloquant qui utilise le principe du protocole de validation à deux phases pour assurer la coordination des processus pendant la réalisation de points de reprise et pendant la reprise sur panne. L'intérêt d'utiliser une validation à deux phases pour réaliser les points de reprise (resp. pour reprendre suite à une panne) est de réaliser l'opération de manière atomique : soit tous les processus concernés le font soit aucun. L'atomicité assure que le protocole peut tolérer des pannes pendant qu'il s'exécute mais également qu'il existe un ensemble de points de reprise qui forment un état global cohérent à tout moment.

Protocole KET pour la réalisation de points de reprise Lorsqu'un processus P décide de se sauvegarder, il réalise un point de reprise local temporaire et envoie des requêtes de sauvegarde à tous les processus Q appartenant à sa cohorte. P est appelé le *processus initiateur*, et la cohorte de P correspond à l'ensemble des processus desquels P à reçu un message depuis sa dernière sauvegarde. En d'autres termes, il s'agit des processus qui devront nécessairement réaliser un point de reprise, à défaut de quoi des incohérences seront créées. Après avoir diffusé la requête de sauvegarde à sa cohorte, le processus P se met en attente d'une réponse à sa requête. Pendant l'attente des réponses, seul le processus applicatif est suspendu : le protocole continue de s'exécuter.

Lorsqu'un processus P' reçoit une requête de sauvegarde RS d'un processus P, il doit répondre au processus P s'il désire participer à la sauvegarde ou pas. S'il ne désire pas participer, il répond négativement. Une telle réponse intervient notamment lorsque P' participe déjà à une opération de réalisation de point de reprise (ou de reprise sur panne). Ce comportement évite de participer à plusieurs opérations concurrentes. Incidemment, une panne est interpretée comme une réponse négative à une demande de sauvegarde ou de reprise. S'il désire répondre affirmativement et que sa cohorte est vide, il répond immédiatement. Dans les autres cas, sa réponse dépendra de la réponse de sa cohorte : il ne répondra affirmativement que si toute sa cohorte y consent. Ainsi, après avoir réalisé un point de reprise temporaire, P' transmet la requête de sauvegarde à sa cohorte et se met en attente d'une réponse. La requête de sauvegarde est ainsi propagée à tous les processus concernés. À cause des dépendances créées par les messages échangés par les processus, il est possible, pour le processus P' de recevoir à nouveau la même requête RS mais de la part d'un autre processus P'' dont il dépend transitivement [4]. Dans ce cas, P' se contente de répondre affirmativement (sans initier de nouvelle propagation). Ce comportement permet d'éviter des propagations infinies.

Les réponses des processus sont propagées de proche en proche jusqu'à atteindre le processus initiateur P. Si tous les processus de la cohorte de P désirent participer à la sauvegarde, le processus P demande à sa cohorte de valider ses points de reprise temporaires : ces derniers deviennent alors des points de reprise permanents. Sinon P annule sa demande de sauvegarde et demande à sa cohorte d'invalider les points de reprise temporaires. La réalisation d'un point de

4. P'' n'est pas nécessairement dans la cohorte de P'

reprise temporaire n'est pas systématique : elle est effective à condition que le plus récent point de reprise permanent disponible introduise des incohérences.

Protocole KET de reprise sur panne La reprise sur panne se déroule de manière similaire à la réalisation de points de reprise. Un processus P, qui redémarre à partir d'un point de reprise, commence par déterminer sa cohorte de reprise et initie une demande de reprise : il envoie à tous les processus de sa cohorte une requête de reprise et se met en attente de leur réponse. Un processus P' qui reçoit une requête de reprise propage cette requête aux processus de sa cohorte et se met en attente d'une réponse. Lorsque P' a reçu une réponse de tous les processus de sa cohorte, il envoie sa réponse au processus qui lui a envoyé la requête de reprise. Les réponses sont ainsi propagées au processus initiateur.

Les processus appartenant à une cohorte de reprise sont des processus qui doivent nécessairement effectuer un retour en arrière afin de garantir la contrainte de cohérence. Pour un processus P' c'est l'ensemble des processus auxquels il a envoyé un message depuis son dernier point de reprise permanent. L'utilisation de cohortes de taille minimale permet d'assurer la participation d'un nombre minimal de processus.

Le protocole *KET* présente des propriétés intéressantes telles que :

1. la garantie d'impliquer un minimum de processus pendant les opérations de reprise sur panne et pendant la réalisation de points de reprise ;
2. la possibilité de supporter des pannes pendant ces deux opérations ;
3. la prise en compte de requêtes concurrentes de réalisation de points de reprise ou de reprise sur panne.

Par ailleurs, n'importe quel processus peut initier une de ces opérations. Ceci vient au prix d'un protocole très complexe et ce, même avec des hypothèses simplificatrices sur la fiabilité des canaux de communication et leur comportement FIFO. Enfin le protocole ne traite pas les messages en transit : ces derniers sont supposés pris en charge par le protocole de transmission.

Cette complexité est sans doute la raison principale de l'inexistence d'implantation du protocole *KET*. Cependant, il existe une implantation d'une version simplifiée appelée *2PCDC* (*Two-phase Commit Distributed Checkpointing*) [103] apparue en 2000. Dans *2PCDC*, l'initiateur des points de reprise est unique et les cohortes sont maximales : elles comprennent tous les processus.

3.4.3 Protocoles à coordination temporelle

Les protocoles à coordination temporelle reposent sur la synchronisation relative d'horloges (ou de minuteurs) pour faciliter la coordination des points des reprise. Les horloges utilisées sont initialement synchronisées et permettent de déclencher périodiquement la réalisation d'un point de reprise sur chacun des processus de l'application distribuée.

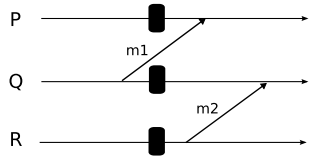


FIGURE 3.7 – Synchronisation temporelle parfaite.

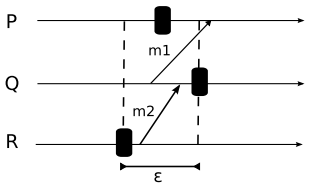


FIGURE 3.8 – Synchronisation temporelle réaliste.

En supposant une synchronisation parfaite entre les processus, ces derniers réaliseraient leur point de reprise exactement au même moment. Cette synchronisation assurerait qu'aucun message orphelin ne soit créé et que l'état global résultant est cohérent. Seuls des messages en transit pourraient être crées. La figure 3.7 illustre ce cas.

Cependant, en réalité, les horloges sur chaque processus peuvent présenter des écarts. Il en résulte que ces écarts peuvent créer des états incohérents. À cause de la synchronisation imparfaite, le processus R peut réaliser son point de reprise plus tôt et le processus Q peut réaliser son point de reprise plus tard de sorte que le message futur $m2$ de la figure 3.7 devienne un message orphelin (cf. Figure 3.8).

Les protocoles de [151, 41] qui ont été proposés au début des années '90 insèrent dans chaque message envoyé, le numéro de séquence i du dernier point de reprise réalisé. Le récepteur du message compare le numéro de séquence i reçu au numéro de séquence j de son dernier point de reprise, et en déduit le type de message. Si $i < j$, il s'agit d'un message en transit que le récepteur intégrera à son dernier point de reprise. Le message sera « rejoué » au processus applicatif correspondant en cas de reprise. Les protocoles reposent sur l'hypothèse que la durée de transmission des messages est bornée et connue pour déterminer le moment à partir duquel il n'y a plus de messages en transit sur le réseau. Lorsque $i > j$, le message deviendra orphelin s'il est livré directement à l'application. Dans [151] le protocole choisit la stratégie qui consiste à refuser le message. Le message finira par être considéré comme perdu par l'émetteur qui le retransmettra. En revanche, dans [41], le protocole force la réalisation d'un point de reprise du processus applicatif avant que ce dernier n'utilise le message. De cette façon, le message devient un message passé.

Les protocoles de [100, 101] ont été introduits dans le milieu des années '90, et diffèrent des deux protocoles précédents dans la manière d'utiliser les informations temporelles pour assurer la cohérence des points de reprise. Par exemple, le protocole décrit dans [100] empêche l'envoi de messages des processus applicatifs pendant un intervalle de temps T' précédant la réalisation d'un point de reprise. La longueur de cet intervalle dépend du délai maximal de transmission d'un message (noté δ), et assure ainsi l'absence de messages en transit pendant la réalisation des points de reprise. Le protocole décrit dans [101] enregistre dans son point de reprise tous les messages qu'il a envoyés mais qui n'ont pas encore été acquittés. Ces messages sont potentiellement des messages en transit : il est possible qu'ils aient été bien reçus mais pas encore acquittés. Ces messages seront rejoués pendant la reprise sur panne : s'ils ont déjà été reçus, leur numéro de séquence permettera au protocole de communication de les détecter comme doublons et de les rejeter.

Pour éviter de créer des messages orphelins, les protocoles de [100, 101] s'abstiennent d'envoyer de nouveaux messages pendant un intervalle de temps T suite à la réalisation du point de reprise. Le calcul de la longueur T de cet intervalle s'appuie sur plusieurs paramètres dont ϵ l'écart maximal que peuvent présenter les horloges et δ le délai maximal de transmission d'un message.

Les protocoles de [101, 151] proposent également un algorithme de resynchronisation d'horloge pour faire face à des écarts qui se créeraient pendant l'exécution. [151] applique l'algorithme de façon périodique pour resynchroniser les horloges. [101] met en place un algorithme pour détecter le moment à partir duquel l'écart entre ses minuteurs devient trop important afin de les resynchroniser. Les auteurs de [101] soutiennent que les re-synchronisations sont rares pour des horloges admettant un *taux de dérive*[5] $\rho = 10^{-6}$ et que l'on trouve dans les stations de travail

5. Le taux de dérive d'une horloge (ou *clock drift rate*) correspond à la différence par unité de temps que

standard.

L'utilisation de tels protocoles permet d'éviter une coordination directe lors de la réalisation de points de reprise : le gain immédiat est la suppression du surcoût occasionné par les communications supplémentaires mises en jeu dans des protocoles tels que le protocole *Sync-aNd-Stop* (cf. Section 3.4.2) qui utilisent une coordination directe. Parmi les protocoles cités précédemment, à notre connaissance, seuls ceux de [101] et [100] ont connu une implantation. En particulier, l'implantation du protocole de [100] a été comparée à une implantation du protocole de coordination non-bloquant décrit dans [44] (cf. Section 3.4.4). Les mesures effectuées sur une partition de 32 nœuds d'une *Connection Machine 5* indiquent que le protocole à coordination temporelle est beaucoup plus performant.

Malgré les performances intéressantes affichées par les protocoles à coordination temporelle, ce type de coordination semble inadapté pour le passage à échelle. Parmi les différentes hypothèses utilisées, celle sur la durée de transmission des messages – supposée bornée et connue – semble difficile à garantir sur des milliers de nœuds.

3.4.4 Protocoles à coordination non-bloquante

Le protocole de Chandy-Lamport (ou *distributed snapshots*) [34] proposé en 1985 est le protocole non-bloquant de réalisation de point de reprise de réference. Dans sa forme initiale, le protocole suppose un nombre fini de processus interconnectés par des canaux (ou *channels*) de communication infaillibles et respectant une politique de transmission FIFO : les messages sortent d'un canal dans le même ordre où ils ont été injectés.

Comme pour le protocole *Sync-aNd-Stop* (Section 3.4.2), il existe un processus responsable de la coordination qu'on appelera le *coordinateur*. Le coordinateur initie une nouvelle phase de sauvegarde en envoyant un message de notification dans tous ses canaux. Après cet envoi, il sauvegarde immédiatement son état. Un processus recevant un message de notification, réalise un point de reprise s'il ne l'a pas encore effectué et envoie une notification dans tous ses canaux. Sinon (s'il a déjà réalisé un point de reprise), il enregistre dans son point de reprise tous les messages qu'il reçoit de canaux dont il n'a pas encore reçu de notification (*i.e.* : il s'agit de messages en retard). L'enregistrement de messages reçus depuis un canal s'interrompt dès la réception d'une notification depuis ce canal. Le point de reprise local à un processus est achevé dès lors qu'il a reçu un message de notification de la part de tous les processus auxquels il est connecté (à partir de ce moment l'enregistrement des messages est interrompu naturellement). Après avoir achevé son point de reprise local, un processus notifie le coordinateur par un message *stop*. Lorsque le coordinateur a reçu un message *stop* de tous les processus, il diffuse un message *end*. Les processus recevant ce message savent que le point de reprise global est terminé. Ils procèdent alors à l'effacement d'anciens points de reprise.

L'hypothèse de canaux de communication fiables et FIFO ainsi que la réalisation de point de reprise immédiatement après la réception de la notification élimine la possibilité de créer des messages orphelins et donc d'aboutir à un état incohérent. De plus, ces mêmes hypothèses garantissent qu'à la réception du message de notification d'un canal, tous les messages en transit, relatifs à un canal de communication, ont été préalablement reçus.

Les hypothèses de canaux de communication fiables et FIFO peuvent être levées avec l'utilisation d'un protocol de transmission fiable comme TCP et en rajoutant l'index du point de reprise dans chaque message échangé [87]. Cet index est incrémenté à chaque fois qu'un nouveau point

présente une horloge par rapport à une horloge de réference idéale. Les horloges quartz dévient d'environ 1s tout les 11 à 12 jours (*i.e.* : 10^{-6} secs/sec)

de reprise global est initié. Enfin, chaque processus P doit informer chacun des autres processus du nombre de messages qu'il leur a envoyé avant de réaliser son point de reprise. Ainsi, chacun des autres processus connaît le nombre de messages en transit qu'il doit attendre de la part de P.

En considérant une application de n processus dans laquelle chaque processus peut communiquer avec tous les autres (topologie complète), le protocole de Chandy-Lamport possède une complexité en nombre de messages en $O(n^2)$. Une telle complexité peut être intolérable pour n très grand. En 2006, [54] propose trois variantes du protocole de Chandy-Lamport qui améliorent la complexité du protocole original en terme de nombre de messages échangés. Les trois protocoles proposés ont des complexités en $O(n^{3/2})$, $O(n \log n \log w)$ et $O(n \log w)$ avec w le nombre moyen de messages en transit par processus. Pour y parvenir, ces protocoles introduisent une topologie virtuelle sous forme de grille ou d'arbre entre les processus.

Il existe dans la littérature quelques implantations de protocoles de coordination non-bloquants [44, 138, 24]. La plus récente [24] date de 2008 et réalise une comparaison expérimentale avec une implantation de protocole de coordination bloquant. Les deux protocoles sont incorporés au sein la bibliothèque de passage de messages MPICH (cf. [60] pour une présentation de MPICH). Les résultats expérimentaux [24] mettent en évidence les faiblesses du protocole de coordination bloquant (voir section 3.4.2) et la meilleure polyvalence du protocole non-bloquant. Notamment, le surcoût introduit par le protocole non bloquant est beaucoup plus faible que le protocole bloquant pour des fréquences de sauvegarde élevées.

Enfin, le protocole de Chandy-Lamport a été adapté pour pouvoir être implanté au niveau de la spécification MPI. Les implantations qui en ont résultées [22, 23, 29] datent de 2003-2004, et présentent l'avantage d'être indépendantes de l'implantation MPI utilisée : les implantations de ces nouveaux protocoles sont donc plus portables que celles de leurs équivalents implantés dans les bibliothèques MPI ou des sous-systèmes réseaux dans les couches plus basses (API sockets, etc.). Ces nouveaux protocoles sont également compatibles pour réaliser des points de reprise distribués au niveau applicatif. [22, 23] utilisent ce nouveau type de protocole pour réaliser des points de reprise distribués au niveau applicatif pour les programmes C/MPI. [29] s'intéresse aux applications FORTRAN/MPI.

3.4.5 Protocoles à coordination fondée sur des modèles de parallélisation

La coordination des processus pour la réalisation de points de reprise formant un état global cohérent peut être simplifiée en étant spécialisée pour le paradigme de programmation parallèle utilisé pour le développement de l'application distribuée, ou pour le paradigme sous-jacent de communication. Les paradigmes de programmation parallèle *Single Program Multiple Data* (SPMD) et *Master-Worker* (MW) (ou *Maître-Travailleur*) mais aussi le paradigme de communication introduit par la technique *Buffered CoScheduling* (BCS) [50] entrent dans cette catégorie.

Dans le paradigme SPMD, plusieurs instances du même programme sont créées et s'exécutent en parallèle mais sur des jeux de donnés différents. Dans ce paradigme de programmation parallèle, de nombreux algorithmes sont constitués de phases de calculs et de phases de communications pendant lesquelles les processus s'échangent les données dont ils ont besoin. Les phases de calculs et les phases de communication se suivent avec une synchronisation plus ou moins stricte selon le modèle suivi par l'algorithme.

Les algorithmes selon le modèle de programmation parallèle BSP (*Bulk Synchronous Parallel*) [155] sont organisés en *super-étapes*. Chaque super-étape comprend une phase de calculs suivie d'une phase de communications avec une synchronisation forte des processus au niveau de la

phase de communications. En effet, cette synchronisation empêche un processus d'entamer la prochaine super-étape avant que tous les autres processus aient fini leurs communications de la super-étape courante. Il existe également des algorithmes similaires mais qui ne suivent pas strictement le modèle BSP. Notamment, ces algorithmes admettent une synchronisation plus relâchée. En effet, chaque processus est libre de continuer dans sa prochaine phase de calculs dès lors qu'il a achevé ses communications de la super-étape courante (*i.e.* : il ne doit pas attendre la terminaison de la phase de communications sur les autres processus). Enfin, il existe les algorithmes selon le modèle PRO (*Parallel Resource-Optimal*) [55], lequel encourage le recouvrement des calculs avec les communications, et autorise donc une synchronisation encore plus relâchée. Toutes ces évolutions du modèle BSP initial visent en fait à augmenter les performances des applications distribuées en supprimant des opérations de synchronisations, en réduisant les temps d'attente des processus les plus rapides, et en réalisant en même temps les calculs et les communications.

Pour réaliser des points de reprise formant un état global cohérent dans une application SPMD, il suffit au programmeur de déclencher un point de reprise dans une section du code applicatif où aucune communication n'a lieu. Comme les processus d'une application SPMD exécutent le même code, ils réaliseront chacun leur point de reprise au même point de l'application et l'état global résultant des points de reprise sera donc cohérent. Une séparation stricte des phases de calculs et des phases de communications sur chaque processus, sans plus de possibilité de recouvrement des calculs et des communications d'un même processus, mènera donc à une réalisation de points de reprise plus simple et plus rapide. Cette approche est facile à mettre en place dans les types d'algorithmes décrits selon le modèle BSP. Elle l'est moins pour les algorithmes selon le modèle PRO.

À titre d'exemple, le système de point de reprise du supercalculateur BlueGene/L d'IBM fournit une primitive `BGLCheckpoint()` [96] que le développeur place dans un endroit de son code applicatif où aucune communication n'a lieu. Lorsqu'un processus appelle cette primitive, un point de reprise du processus est réalisé. Mais cette primitive est typiquement appelée après une barrière de synchronisation (comme `MPI_Barrier`) si l'algorithme ne sépare pas strictement de lui même les communications et les calculs. D'autres approches utilisant cette spécificité du modèle SPMD seront présentées à la section 3.8. Au chapitre 5 nous présentons la solution que nous avons adoptée.

Le paradigme MW permet également de simplifier la réalisation d'états globaux cohérents. Dans ce paradigme un processus maître distribue des tâches à traiter à des processus travailleurs. Ces derniers ne communiquent pas entre eux. Ils communiquent uniquement avec le processus maître pour recevoir des tâches et lui renvoyer le résultat de ces dernières. Cette organisation de l'application permet de mettre en place un schéma de sauvegarde très simple qui consiste simplement à sauver le maître et la liste de ses tâches. Celles qui étaient en train d'être traitées pendant la réalisation du point de reprise seront redistribuées à la reprise. D'autres schémas sont possibles pour les applications suivant le paradigme MW : ils seront examinés en détail au chapitre 6.

Les protocoles décrits dans le paragraphe précédent reposent sur le paradigme de programmation parallèle et la connaissance de la sémantique applicative. Alternativement, comme le démontre la technique de *BCS*, il est également possible de se fonder sur la connaissance d'un paradigme de communication particulier, et uniquement sur lui, pour faciliter la réalisation de points de reprise cohérents. La technique de *BCS* permet d'implanter des bibliothèques de passage par message dans lesquelles le temps d'exécution de l'application est discrétisé en intervalles fixes de quelques centaines de microsecondes pendant lesquels sont plannifiées les communications. La discrétisation s'accomplit grâce à des synchronisations fréquentes. Une conséquence intéressante

de cette approche est qu'à la fin de chaque intervalle aucun message ne circule dans le réseau. La réalisation de points de reprise s'en trouve facilitée dans la mesure où la réalisation de points de reprise à l'issue de ces intervalles forment un état global cohérent.

3.5 Protocoles de points de reprise à enregistrement de messages

Avec les protocoles à enregistrement de messages, l'état d'un processus peut être recréé à partir d'un point de reprise et de messages qui ont été enregistrés. La principale différence avec les protocoles par retour arrière fondés uniquement sur les points de reprise est la possibilité d'éviter un retour arrière de tous les processus : ainsi, en cas de panne, l'application redémarre à partir d'un état plus récent que si elle utilisait un protocole de points de reprise.

Une hypothèse courante dans ces protocoles est celle de l'exécution déterministe par morceau (ou *piecewise deterministic*) [146] selon laquelle l'exécution d'un processus correspond à une séquence d'intervalles déterministes délimités par l'arrivée d'évènements non déterministes. En l'occurrence, ici, l'évènement non déterministe qui nous intéresse est la réception de messages et leur ordre de réception. En enregistrant les messages et leur ordre d'arrivée, un processus peut être rétabli en rejouant les messages dans le même ordre où ils ont été reçus. On associe donc aux messages enregistrés un numéro de séquence de réception (ou NSR) et/ou un numéro de séquence d'envoi (ou NSE). Les données d'un message ainsi que les informations qui permettent de le rejouer dans le bon ordre à la reprise forment ce qui est communément appelé un *déterminant* [43].

Dans ces protocoles, on distingue lors de la réception d'un message le moment où ce dernier est enregistré et le moment où il est livré à l'application. En particulier, la réception d'un message ne signifie pas qu'il est immédiatement accessible par le processus applicatif. Les protocoles par enregistrement de messages peuvent être regroupés selon la stratégie qu'ils adoptent pour enregistrer les messages : *pessimiste*, *optimiste* et *causal*.

3.5.1 Enregistrement pessimiste de messages

Les protocoles pessimistes d'enregistrement de messages enregistrent systématiquement les messages avant que ces derniers n'aient pu influencer l'état des processus. Ce type d'enregistrement est dit *synchrone* et peut avoir lieu, selon le protocole, sur l'émetteur ou le récepteur du message. Cette stratégie est qualifiée de pessimiste car elle considère que la probabilité d'occurrence d'une panne est très élevée ; notamment entre le moment où le message est enregistré et le moment où il est transmis au processus de l'application.

3.5.1.1 Enregistrement chez le récepteur

La version la plus simple de ce type de protocoles est sans doute celle où l'enregistrement est effectué par le récepteur : un message reçu n'est livré à l'application qu'après avoir été enregistré avec succès par le processus récepteur sur support stable. La reprise sur panne d'un processus d'une application utilisant ce protocole s'effectue comme suit : le processus est redémarré à partir de son point de reprise le plus récent et les messages qu'il a reçus après le point de reprise lui sont rejoués dans le même ordre.

Ce protocole pessimiste comprend de nombreux avantages que l'on retrouve listés dans [43]. Pour ce qui nous concerne, nous pouvons citer la simplicité de la procédure de reprise sur panne et celle de gestion des points de reprise et des enregistrements (ou déterminants) associés. En

effet, un processus qui tombe en panne peut reprendre son exécution de manière autonome sans impliquer les autres processus. Cependant, ces derniers devront attendre la fin de la reprise sur panne du processus défaillant avant de pouvoir à nouveau intéragir avec lui. Par ailleurs, le protocole garantit qu'un processus qui tombe en panne pourra toujours effectuer une reprise sur panne à partir du dernier point de reprise et des enregistrements associés. Les points de reprise plus anciens peuvent donc être effacés sans crainte.

Le prix à payer pour ces avantages est une baisse de performance très importante qui découle de l'enregistrement synchrone des messages sur support stable. Une manière d'y remédier est d'utiliser un protocole dans lequel les messages sont enregistrés sur l'émetteur en mémoire volatile.

3.5.1.2 Enregistrement chez l'émetteur

Un des premiers protocoles à enregistrement pessimiste de messages à l'émetteur a été proposé en 1987 dans [72]. Dans ce protocole, le récepteur renvoie à l'émetteur un acquittement contenant le numéro de séquence de réception (ou NSR) du message reçu. Les messages pour lesquels l'émetteur connaît le numéro de séquence d'envoi (ou NSE) et le NSR sont dits *entièrement enregistrés*. Tant que le récepteur n'a pas reçu d'acquittement de la part de l'émetteur confirmant l'enregistrement du message, il est interdit au récepteur d'envoyer de nouveaux messages. Ce comportement est crucial afin de garantir que les autres processus ne deviennent dépendants d'un évènement qui n'a pas encore été enregistré dans un endroit sûr.

Lorsqu'un processus tombe en panne, il reprend son exécution à partir de son dernier point de reprise et envoie à tous les autres processus le NSE du dernier message qu'il a reçu : le NSE du dernier message reçu fait partie du point de reprise. Les processus émetteurs en déduisent les messages qu'ils doivent ré-envoyer. Les messages sont ré-envoyés avec leur NSR afin que le processus qui récupère de la panne sache dans quel ordre rejouer les messages. Des messages peuvent avoir été partiellement enregistrés parce que le récepteur est tombé en panne avant leur réception ou avant d'avoir pu les acquitter. Les messages partiellement enregistrés sont envoyés après l'envoi des messages entièrement enregistrés et sont rejoués dans leur nouvel ordre d'arrivée. L'ordre dans lequel sont reçus ces messages pendant la reprise sur panne importe peu. En effet, étant donné que le récepteur ne peux pas envoyer de messages avant d'avoir acquittés les messages qu'il a reçus, l'état d'aucun autre processus ne dépend de ces messages et l'ordre de réception de ces messages n'affecte aucunement la correction (de l'application).

Le protocole suppose que chaque processus effectue de manière périodique un point de reprise local. Les déterminants sont alors écrits sur support de stockage stable avec le point de reprise. Bien que ce protocole permette d'éviter le surcoût lié à l'accès au support stable, il ne tolère qu'une unique panne de processus. Ceci découle en partie du fait que pour chaque processus, les informations sur l'ordre de réception des messages sont distribuées. En conséquence, si plus d'un processus tombe en panne il devient impossible de connaître quels messages devront être rejoués par quels processus et dans quel ordre.

MPICH-V2 [16] est un protocole pessimiste enregistrant les messages chez l'émetteur. Par rapport au protocole de [72], *MPICH-V2* introduit entre autres des enregistreurs d'évènements. Un enregistreur d'évènement correspond à un processus situé sur une machine supposée fiable. Le rôle de ces enregistreurs est de sauver pour chacun des processus des informations sur les messages qu'ils reçoivent. Ces informations comprennent notamment l'identifiant de l'émetteur, le NSE et le NSR du message reçu et sont utilisées pendant les reprises sur pannes pour connaître quels sont les messages qui doivent être rejoués, à quel processus les demander et aussi dans quel ordre les rejouer.

Par l'intermédiaire des enregistreurs, les informations sur les messages reçus par un processus sont découplées des processus émetteurs et elles sont centralisées. Il s'ensuit que, même si plusieurs processus de calcul tombent en pannes, les informations sur les dépendances entre les messages restent accessibles. *MPICH-V2* repose alors sur la ré-exécution des processus fautifs pour regénérer les messages perdus à cause de pannes.

[16] évalue les performances du protocole *MPICH-V2* qui a été implanté au sein de la bibliothèque de communication MPICH. L'utilisation des enregistreurs de messages introduit une latence supplémentaire qui rend *MPICH-V2* peu performant pour les applications mettant en jeu beaucoup d'échanges de petits messages. Dans le reste des cas *MPICH-V2* affiche des performances comparables à des exécutions n'utilisant pas le protocole.

3.5.2 Enregistrement optimiste de messages

Les procotoles à enregistrement de messages optimistes [146] tentent de réduire le surcoût introduit par les protocoles pessimistes en écrivant les déterminants sur support de stockage stable de manière asynchrone. Ces protocoles sont qualifiés d'optimistes car ils se fondent sur l'hypothèse optimiste que l'enregistrement des messages pourra avoir lieu avant l'occurrence d'une panne. Les déterminants sont donc conservés en mémoire volatile et sont écrits sur le support de stockage stable périodiquement (par exemple). Grâce à cette hypothèse, l'application n'est plus obligée de rester bloquée jusqu'à ce que les messages aient été enregistrés et conduit potentiellement à des exécutions plus rapides en l'absence de pannes. Malheureusement, cette hypothèse n'est pas toujours vérifiée et, en cas de pannes, les informations sur le contenu et l'ordre des messages sont perdues. Si le processus qui tombe en panne a envoyé des messages à d'autres processus ces derniers deviennent orphelins et doivent aussi effectuer un retour arrière pendant la reprise sur panne. Les protocoles optimistes peuvent donc entraîner plusieurs processus à effectuer un retour arrière. La détermination de la ligne de reprise requiert un protocole complexe de reprise.

3.5.3 Enregistrement causal de messages

Les protocoles par enregistrement de messages dits *causaux* essaient de conserver les avantages des protocoles pessimistes et optimistes : d'une part ils ne nécessitent aucun bloquage des processus de l'application distribuée en l'absence de pannes et d'autre part ils garantissent lorsqu'un processus tombe en panne, que la reprise sur panne pourra s'effectuer à partir du dernier point de reprise de ce processus.

3.5.4 Bilan des implantations des protocoles à enregistrement de messages

Malgré un nombre très important de travaux théoriques sur ces protocoles par enregistrement de message, il y a finalement eu peu d'implantations, notamment pour les protocoles optimistes et causaux. Parmi les implantations on peut citer celles réalisées avec *EGIDA* [122] en 1999. Il s'agit d'un toolkit destiné à fournir un environnement uniforme pour l'implantation et la comparaison de protocoles de reprise par enregistrement de messages. L'implantation de ces protocoles s'avère très complexe. Dans le domaine commercial, le seul cas d'implantation que l'on connaisse est celui de protocoles pessimistes dans des systèmes de télécommunications [66] en 1995. Enfin, nous pouvons citer le projet MPICH-V du LRI (Laboratoire de Recherche en Informatique) de l'Université de Paris Sud. Au cours de ce projet qui s'est déroulé de 2002 à 2007, plusieurs protocoles à enregistrement de messages ont été conçus et implantés au sein de la bibliothèque

de passage de message MPICH. MPICH-V1 [15] et MPICH-V2[6] [16] sont deux protocoles à enregistrement pessimiste de messages. MPICH-V propose également MPICH-VCausal [83] qui est un protocole à enregistrement causal de messages.

3.6 Classification des réalisations en tolérance aux pannes

Dans cette section nous nous intéressons à ce qui a été accompli en termes de réalisations en tolérances pannes. Nous portons un intérêt particulier aux efforts pour apporter de la tolérance aux pannes aux applications utilisant la bibliothèque de passage de message MPI [141]. Cette dernière s'est imposée comme le standard *de facto* pour la distribution d'applications non trivialement parallélisables (ou *embarrassingly parallel applications*) sur des grappes de PCs.

Dans le cadre des sections 3.3, 3.4 et 3.5, nous avons présenté et discuté de nombreux protocoles de reprise par retour arrière. Ces protocoles sont effectivement cruciaux dans la mise en œuvre de solutions de tolérance aux pannes et peuvent constituer un critère de classification des réalisations en tolérance aux pannes. Néanmoins, pour cette classification il nous a semblé important de tenir compte également du *niveau d'implantation* de la solution de tolérance aux pannes (ou *Sytème de réalisation de Points de Reprise Distribué* (SPRD)). Le critère de niveau d'implantation s'applique aussi bien au protocole de coordination qu'au *Sytème de réalisation de Points de Reprise Séquentiel* (SPRS). Ces deux composantes constituent le SPRD. Les deux peuvent être implantés au niveau applicatif ou au niveau système (ou middleware). Souvent les deux sont implantés au même niveau mais ceci n'est pas une obligation. Nous avons par exemple des cas où la coordination au niveau applicatif est couplée à un SPRS de niveau système.

Les critères de classification comprennent :

- le *niveau de réalisation du point de reprise* : applicatif ou système. Ces niveaux se distinguent dans la manière de sauver et de restaurer un processus. Au niveau applicatif, l'application doit être modifiée dans son code source afin de pouvoir sauver et restaurer son contexte. Au niveau système, c'est le processus qui est sauvé et restauré. Ceci comprend la sauvegarde de la pile d'exécution, de registres, etc.
- l'*origine de l'ordre de réalisation du point de reprise*. Le point de reprise peut être initié par l'application ou par le système. Dans le premier cas, le code de l'application est modifié pour déclencher des points de reprise à différents points de l'exécution. Dans le second cas, c'est un évènement extérieur à l'application qui décide.
- la *capacité à être piloté*. Ceci désigne la capacité à pouvoir intéragir avec le protocole de réalisation de points de reprise dans le but de déclencher à la demande la réalisation d'un point de reprise de l'application (avant une opération de sauvegarde, par exemple), de modifier la fréquence de réalisation de points de reprise ou tout autre paramètre relatif au fonctionnement de la tolérance aux pannes.
- la *portabilité* du SPRD : il s'agit de la capacité à fonctionner sur plusieurs systèmes différents ;
- la *portabilité* des points de reprise réalisés : c'est la capacité pour un point de reprise réalisé sur un architecture donnée à pouvoir être utilisé pour reprendre l'application sur une autre architecture.
- la *transparence* de la solution pour l'utilisateur.

Le choix du niveau d'implantation conditionne très souvent les caractéristiques du SPRD résultant. Une implantation au niveau système est caractérisée par une très bonne transparence vis-à-vis de l'utilisateur. L'application peut facilement être préemptée car les points de reprise

6. MPICH-V2 a été détaillé à la section 3.5.1.2

peuvent être réalisés à n'importe quel point de son exécution. En contre partie, l'approche n'est pas très portable : une migration n'est envisageable que sur des systèmes strictement identiques [7].

En effet, une architecture différente ou un système d'exploitation de version légèrement différente peuvent suffir pour faire échouer la manœuvre. Enfin, la réalisation de points de reprise de niveau système résulte dans des tailles de points de reprise importantes : tous les octets formant le processus sont sauvés.

Au niveau applicatif, nous observons une tendance inversée. Les points de reprise sont très portables et leur volumes peuvent être minimisés. Le SPRS est aussi plus portable étant donné qu'il dépend moins d'éléments non portables du système hôte. En revanche, l'approche est moins transparente car elle nécessite une intervention du programmeur et des modifications dans le code source de l'application. La capacité du SPRD à préempter l'application est également très limitée car au niveau applicatif, les points de reprise ne peuvent être réalisés qu'aux seuls endroits marqués dans le code source.

La section 3.7 dresse le bilan des solutions implantées au niveau système, et la section 3.8 étudie celles de niveau applicatif.

3.7 Réalisations au niveau système

Les solutions implantées au niveau système peuvent être réparties selon qu'elles sont implantées en *espace noyau* ou en *espace utilisateur*. Les solutions en espace noyau bénéficient d'une transparence parfaite vis-à-vis de l'utilisateur. Elles ne nécessitent aucune modification du code source. Néanmoins elles sont très sensibles à des modifications du noyau. Le coût de leur maintenance est très élevé ce qui limite leur utilisation à un système unique. Les solutions en espace utilisateur reposent sur de la virtualisation. La virtualisation peut être légère et portée sur des APIs telles que l'API des sockets, et d'autres appels systèmes [4, 127].

Ensuite il existe des approches spécifiques à MPI. Ces approches consistent à rendre une implantation MPI tolérante aux pannes.

BGL_Checkpointer [96] est la solution de réalisation de point de reprise fournie avec les supercalculateurs BlueGene d'IBM. Il s'agit d'une bibliothèque que le programmeur utilise dans son application pour marquer les endroits où des points de reprise peuvent être réalisés. Le programmeur doit s'assurer de l'absence de messages en transit au moment où cet appel est réalisé. Le déclenchement de la réalisation des points de reprise est donc interne à l'application [8]. Il revient au programmeur de rajouter le code nécessaire pour la périodicité des sauvegardes.

CLIP (Checkpointing Libraries for the Intel Paragon) [36] était une bibliothèque qui permettait de réaliser des points de reprise d'applications distribuées par passage de messages sur des machines Intel Paragon. La réalisation des points de reprise était conditionnée par la présence d'une instruction de checkpoint dans le code source de l'application. Cette instruction déclenchait un point de reprise lorsqu'un intervalle de temps minimum s'était écoulé depuis la dernière sauvegarde. *CLIP* utilisait un protocole de coordination de type *sync-and-stop* (cf. Section 3.4.2) et réalisait des points de reprise de niveau système avec la bibliothèque *libckpt* [114].

7. C'est vraisemblablement le cas dans un système à image unique (ou *Single System Image*) comme Kerrighed [74], MOSIX [9] ou plus récemment XtreemOS [162].

8. Dans la littérature nous retrouvons les termes de *application-initiated*, *application-triggered* ou *application-driven*.

CoCheck [142] a été une des premières tentatives pour intégrer un protocole de réalisation de points de reprise distribué au sein d'une bibliothèque MPI. Le protocole de coordination utilisé est bloquant et il est implanté au niveau de la spécification MPI. Donc, le protocole se situe au-dessus de l'implantation MPI (et non au niveau de l'API des sockets). Cette approche permet à CoCheck de limiter les modifications à apporter à l'implantation MPI utilisée. En revanche, avec une complexité en nombre de messages échangés en $O(n^2)$, le protocole s'avère être assez coûteux. Enfin, CoCheck utilise le SPRS de l'environnement Condor : il s'agit d'un SPRS de niveau système (en espace utilisateur) qui a été utilisé dans de nombreux autres travaux de recherche sur la tolérance aux pannes.

MPICH-V [17] est un framework générique conçu et mis en œuvre à l'université de Paris Sud dans le cadre du projet éponyme. *MPICH-V* repose sur la bibliothèque *MPICH* [95] et permet de fournir une solution de tolérance aux pannes automatique (semi-)transparente aux applications qui l'utilisent. Contrairement à d'autres efforts pour ajouter des protocoles de réalisation de points de reprise à MPI, MPICH-V propose une architecture dans laquelle des nœuds jouent le rôle de serveurs de stockage des points de reprise (ou *checkpoint servers*). Ces nœuds sont reservés en plus des nœuds de calculs. *MPICH-V* offre le choix entre plusieurs protocoles de reprise par retour arrière :

- *MPICH-V1*[15] combine un protocole de réalisation de points de reprise non coordonné avec un protocole d'enregistrement de message pessimiste. La particularité du protocole d'enregitrement est de faire transiter les messages par des nœuds spéciaux appelés *canaux mémoire* (*memory channels*) ; chaque message m_{ij} d'un nœud i à un nœud j transite par un canal mémoire. Ce dernier enregistre le message m_{ij} sur son disque dur local lorsqu'il le reçoit de i et avant de le transmettre à j. C'est ainsi qu'est réalisé l'enregistrement pessimiste des messages. Le protocole est similaire à celui décrit dans [] et affiche une grande latence. En contre partie, il permet de récuperer de la panne de n'importe quel nœud de calcul sans avoir besoin de réaliser de retour arrière d'autres nœuds ;
- *MPICH-V2* [16] conserve un protocole de réalisation de points de reprise non coordonné mais le combine avec un protocole pessimiste d'enregistrement de message chez le récepteur. Les canaux mémoires ne sont plus utilisés et sont remplacés par des enregistreurs d'évènements : il s'agit de processus situés sur des nœuds supposés fiables.
- *MPICH-V/Causal* [83] conserve un protocole de réalisation de points de reprise non coordonné mais combiné avec un protocole d'enregistrement de message causal. Les enregistreurs d'événements distants sont également conservés ;
- *MPICH-V/CL* [24] implante le célèbre protocole de coordination non-bloquant (*Distributed snapshots*) de Chandy-Lamport [34] ;
- *MPICH-PCL* [24] est une implantation d'un protocole de coordination bloquant de réalisation de points de reprise.

LAM/MPI [129] est parmi les premières implantations MPI à intégrer un framework modulaire pour la tolérance aux pannes. La tolérance aux pannes est de niveau système et fondée sur un protocole bloquant de réalisation de points de reprise. La modularité permet de supporter et donc d'utiliser plusieurs SPRS. LAM/MPI fonctionne bien avec le SPRS BLCR de niveau système et en espace noyau de l'université de Berkeley.

Open MPI [67] est l'implantation MPI qui a succedé à LAM/MPI. Open MPI intègre également un framework de tolérance aux pannes inspiré de celui de LAM/MPI et donne la possibilité

d'être associé à différents SPRS. Pour l'instant, tout comme LAM/MPI, Open MPI utilise BLCR.

EGIDA [122] est un toolkit orienté objet qui permet de spécifier des protocoles de reprise par retour arrière arbitraires. Ce toolkit s'appuie sur l'observation que ces protocoles partagent une structure événementielle commune. Ceci permet de spécifier les protocoles de manière modulaire et favorise la réutilisation de modules existants pour réaliser un prototypage rapide de nouveaux protocoles. Notamment, *EGIDA* a permis d'analyser expérimentalement plusieurs protocoles de réalisations de points de reprise induites par les communications (*Communication-Induced Checkpointing (CIC)*) sur les *NAS Parallel Benchmarks* (NPB) [3]. NPB est un ensemble d'applications qui sont issues d'applications réelles de calcul de dynamique des fluides et qui sont destinées à aider l'évaluation de machines parallèles. De nouveaux protocoles dits *hybrides* ont pu être expérimentés [121]. Leur nom dérive du fait que ces protocoles essaient de combiner le faible surcoût à l'exécution des protocoles à enregistrement de messages sur l'émetteur et le faible surcoût à la reprise des protocoles à enregistrement de messages sur le récepteur.

EGIDA est integré à MPICH-1 et apporte une tolérance aux pannes transparente aux applications MPI. Il est conçu pour supporter uniquement les pannes de processus temporaires. EGIDA intègre son propre SPRS et un *watchdog* a été mis en place pour relancer automatiquement un processus défaillant. Le *watchdog* est un mécanisme dans lequel le processus supervisé notifie régulièrement le chien de garde qu'il est en forme. En cas de manquement d'une échéance : si aucune notification n'a été reçue à l'expiration d'un temporisateur (*timer*), le processus supervisé est considéré comme défaillant et la phase de reprise est enclenchée.

DejaVu [127] est un système de réalisation de points de reprise distribués en espace utilisateur initialement développé à *Virginia Tech* aux États-unis. Le code source est fermé et tout porte à croire que le logiciel fait parti de l'offre de la compagnie *Librato* [84]. DejaVu n'est pas la première approche en espace utilisateur mais affiche un meilleur niveau de transparence par rapport à des solutions antérieures. Cette transparence est réalisée grâce à une virtualisation importante au niveau des APIs proposées par le noyau du système d'exploitation. Ceci comprend entre autres l'API des sockets et celle des entrées/sorties.

Pour ce qui est du protocole de réalisation de point de reprise, DejaVu repose sur un protocole appelé *protocole d'enregistrement en ligne* : les messages sont enregistrés chez l'émetteur jusqu'à recevoir un acquittement du récepteur. L'acquittement confirme la bonne réception et provoque l'invalidation du message de l'espace d'enregistrement de l'émetteur. Les messages sont par ailleurs séquencés de manière à détecter de potentielles pertes [9] ou des duplications et à conserver l'ordre d'envoi des messages. Cette approche permet à DejaVu de maintenir une image de l'état du réseau. En temps normal ce type d'information est caché au sein du protocole de transmission fiable et reste inaccessible (et donc incontrôlable) à l'extérieur du noyau du système d'exploitation.

Le protocole de réalisation de point de reprise est une variante du protocole Sync-aNd-Stop (cf. Section 3.4.2, p. 33). Le processus initiateur diffuse la demande de réalisation de point de reprise à tous les autres processus. Ces derniers suspendent les communications en cours, réalisent un point de reprise local et informent le processus initiateur du succès de l'opération. Après réception de tous les messages des autres processus, le processus initiateur leur diffuse la confirmation de réussite de l'opération de réalisation du point de reprise global. Suite à cette confirmation chaque processus reprend son cours normal. La principale différence avec le protocole Sync-aNd-Stop est que le contrôle sur l'envoi et la réception que se créé DejaVu lui permet

9. Utilisation d'un mécanisme classique d'expiration de temporisateur.

de bloquer uniquement les communications : les processus de calcul continuent leur exécution et les communications peuvent être différées le temps de la coordination.

L'approche de se fonder sur son propre protocole de transmission n'est pas neuve en soi et a déjà été utilisée par le passé dans d'autres implantations de protocoles de reprise par retour arrière (*e.g.* : [101]). Enfin, DejaVu réalise les points de reprise de manière incrémentale.

DMTCP (Distributed MultiThreaded CheckPointing) [4, 42] DMTCP est tout comme DejaVu un SPRD en espace utilisateur qui propose un niveau de transparence comparable à celui de DejaVu. DMTCP est développé conjointement par le *MIT* et la *Northeastern University*. À la différence de DejaVu, DMTCP est open source. Le protocole de réalisation de point de reprise de DMTCP est également bloquant mais ne nécessite pas de contrôle aussi fin que celui de DejaVu sur les messages de communication. Enfin, DMTCP repose sur le SPRS MTCP [123], lui-même en espace utilisateur, et qui supporte les processus multi-threadés.

Starfish [1] est un système visant à permettre et à faciliter l'exécution d'applications MPI-2 sur des grappes de PCs non nécessairement homogènes. Starfish associe à chaque processus applicatif un processus démon. Ces derniers forment des groupes légers qui communiquent entre eux par des communications de groupe (assurées par *Ensemble* [156]). Les démons de Starfish assurent des tâches de tolérance aux pannes telles que la détection de nœuds (ou processus défaillants) ainsi que l'ajout (ou la suppression) dynamique de nœuds de calcul. Une originalité de Starfish est sa portabilité qui est atteinte à travers l'utilisation de la machine virtuelle d'OCaml : au lieu de sauver et restaurer du code binaire natif, Starfish sauve et restaure du *bytecode* de la machine virtuelle OCaml. L'utilisation d'une machine virtuelle pour atteindre la portabilité est sans doute aussi le point faible de l'approche étant de donné que le code est à ce moment interprété et par conséquent plus lent.

Buffered CoScheduled MPI [49, 111] est une implantation de la spécification MPI dans laquelle le temps d'exécution de l'application est discretisé en intervalles fixes (quelques centaines de microsecondes) pendant lesquels sont plannifiées les communications. Un aspect important qui découle de cette approche est l'absence de messages sur le réseau à la fin de chaque intervalle. Dans un contexte de tolérance aux pannes cela facilite la réalisation de points de reprise distribué. En effet, ce comportement est intéressant dans la mesure où la fin de chaque intervalle constitue un état global cohérent. Des points de reprise réalisés à ces instants sont donc automatiquement cohérents entre eux.

***AMPI* (Adaptive MPI)** [64] est une implantation MPI au-dessus de Charm++ [73]. Charm++ est un système dont la principale caractéristique est de reposer sur une *virtualisation des processeurs* dont hérite AMPI directement. Cette technique associe un processus MPI à un processeur virtuel. Les processeurs virtuels sont ensuite associés à des processeurs physiques. Les processeurs virtuels sont d'habitude beaucoup plus nombreux que les processeurs physiques. Ainsi plusieurs processeurs virtuels sont associés à un processeur physique. Ce phénomène de surdécomposition (ou *overdecomposition*) qui en résulte est très exploité par AMPI (via Charm++). Il permet de réaliser naturellement des recouvrements entre les calculs et les communications des processeurs virtuels affectés à un même processeur physique. AMPI l'exploite également pour réaliser un équilibrage de charge dynamique : un processeur physique surchargé verra une partie de ses processeurs virtuels réaffectés à d'autres processeurs physiques. La réaffectation est accomplie par de la *migration de processus*.

De nombreux protocoles aussi bien de réalisation de points de reprise [63, 163] que d'enregistrement de messages [32, 33] ont été implantés sous Charm++. [33] propose un protocole pessimiste d'enregistrement de messages à l'émetteur dont la particularité est d'utiliser les processus non défaillants pour accélerer la reprise : suite à la panne d'un processeur physique, les processeurs virtuels de ce dernier sont répartis sur plusieurs autres processeurs physiques de manière à accélerer la reprise. Ce type d'approche est rendu possible par la surdécomposition et le mécanisme de migration inhérents à Charm++.

3.8 Réalisations au niveau applicatif

Les solutions au niveau applicatif se distinguent d'une part par la technique utilisée pour réaliser les points de reprise : comment sauver et restaurer correctement le contexte de l'application. D'autre part, elles se distinguent aussi par la coordination adoptée.

Trois types de coordinations se retrouvent au niveau applicatif : la première est celle qui repose sur un itérateur de boucle, la seconde est déclenchée par un temporisateur et la troisième repose sur le nombre de passages par des emplacements de réalisation de points de reprise.

3.8.1 Réalisations à la charge du programmeur

Les premières réalisations auxquelles nous nous intéressons requièrent une implication importante de la part du programmeur dans la mesure où se dernier est livré à lui-même pour transformer le code son application en un code tolérant aux pannes : à savoir un code capable de sauver le contexte applicatif au moment de la réalisation du point de reprise et de le restaurer au moment de la reprise.

SRS (Stop Restart Software) [153] est un framework pour faciliter le développement d'applications MPI C ou FORTRAN qui sont *malléables*[10] et capables de migrer. Bien que son objectif ne soit pas la tolérance aux pannes, *SRS* accompli la malléabilité et la migration d'une application par l'intermédiaire de la réalisation de points de reprise distribués. L'application est donc sauvée sur disque et puis relancée sur un autre ensemble de machines dont le nombre est possiblement différent.

La réalisation de points de reprise et la relance à partir de points de reprise reposent sur le programmeur. Ce dernier définit le contenu des points de reprise grâce à une bibliothèque de réalisation de points de reprise fournie par *SRS*. La restitution du contexte d'exécution au moment de la reprise est également accomplie par le programmeur en utilisant des instructions conditionnelles appropriées.

TCS CPR (Terascale Computing System CheckPoint and Restart library) [145] est une bibliothèque conçue et mise en œuvre au centre de calcul intensif de Pittsburgh (Pittsburgh Supercomputing Center (PSC)) pour subvenir à leur besoins en matière de tolérance aux pannes. La bibliothèque a pour but de fournir les moyens au programmeur de transformer ses applications écrites en C/C++ ou FORTRAN en apportant une interface commune de sauvegarde/restauration des données de l'application. Ainsi la tolérance aux pannes est assurée par le programmeur de manière similaire au développement avec le framework *SRS*.

10. La malléabilité désigne la capacité de pouvoir dynamiquement modifier le nombre d'entités de calcul d'une application.

FT-MPI [47, 52, 48] est une implantation de la spécification MPI-1.2 qui intègre une sémantique de tolérance aux pannes plus riche que celle fournie par les spécifications MPI. Actuellement, la spécification MPI offre au programmeur deux possibilités pour traiter les pannes. La première, qui est le comportement par défaut, consiste à immédiatement interrompre toute l'application. La seconde, un peu plus flexible, redonne la main à l'application utilisateur sans toutesfois garantir la possibilité d'effectuer de nouvelles communications. Le but principal de la seconde est plutôt de permettre à l'application de quitter proprement (*e.g.* : fermeture de fichiers ouverts, ...)

Suite à une panne d'un processus, la bibliothèque MPI est laissée dans un état incohérent qui appelle à ce que des mesures soient prises afin que l'application soit remise dans un état cohérent. L'incohérence concerne d'une part le statut des objets MPI ; et d'autre part, les éventuelles communications en cours (aussi bien simples que collectives).

Les objets MPI concernés sont ceux qui disposent d'un état global : typiquement les communicateurs. Ces derniers passent automatiquement dans un état défaillant suite à la détection d'une panne. FT-MPI offre alors la possibilité de remplacer le processus défaillant ou de ne rien faire. Dans le premier cas, FT-MPI permet de reconstruire le communicateur (`FTMPI_COMM_MODE_REBUILD`) alors que dans le second, FT-MPI permet de laisser la place du processus défaillant vacante (`FTMPI_COMM_MODE_BLANK`) ou de réduire la taille du communicateur (`FTMPI_COMM_MODE_SHRINK`).

FT-MPI offre deux modes pour traiter les messages suite à l'occurrence d'une panne. Le premier annule tous les messages en transit. Ce mode est adapté pour les applications où la reprise signifie une relance de tous les processus à partir d'un état cohérent. Le second mode complète le transfert de tous les messages sauf ceux à destination et en provenance de processus défaillants.

Avec FT-MPI, le programmeur se voit offrir les moyens d'écrire des applications MPI tolérantes aux pannes de processus. Le programmeur dispose donc d'un outil puissant mais dans lequel de nombreuses tâches relatives à la tolérance aux pannes restent à sa charge. L'ajout de la tolérance aux pannes est donc manuel et laisse la porte ouverte à de nombreuses erreurs lors de l'écriture du code.

3.8.2 Réalisations fondées sur un framework

Plusieurs réalisations de tolérance aux pannes fondées sur des frameworks ont vu le jour. Nous en exposons quelques-unes ici. La plupart concernent les applications SPMD.

PUL-RD [140] était une bibliothèque dont l'objectif était de faciliter l'écriture d'applications parallèles fondées sur une décomposition régulière de domaine. Le domaine correspondait à un tableau (possiblement à N dimensions). Le partage des données du tableau était réalisé à travers la spécification d'un opérateur « stencil » par le programmeur. La tolérance aux pannes était considérée par les auteurs comme entièrement transparente dans la mesure où l'application codée par le programmeur bénéficiait immédiatement d'une tolérance aux pannes. Une autre caractéristique mise en avant était la possibilité de reconfigurer l'application à partir d'un point de reprise distribué afin qu'elle continue son exécution sur un nombre différent de processus (*i.e.* : malléabilité).

***DOME* (*Distributed Object Migratable Environment*)** [5] était un framework pour le développement d'applications SPMD en C++ dans des environnements hétérogènes. DOME reposait sur PVM pour la distribution, la détection de pannes et la relance. Pour tenir compte de l'hétérogénéité, DOME réalisait de la tolérance aux pannes au niveau applicatif [11]. DOME

fournissait ses propres types qui étaient les seuls à pouvoir être sauvés et restaurés ce qui peut être contraignant pour un programmeur dans la mesure où il ne peut utiliser la structure de données de son choix pour ses calculs. DOME proposait deux modes de sauvegarde/reprise. Le premier se fondait sur la ré-exécution et la structure SPMD de l'application. Cependant, il ne permettait pas d'éviter de réexécuter automatiquement tous calculs déjà effectués. Le second mode donnait la possibilité de placer des emplacements de réalisation de points de reprise au sein même de fonctions. La transformation du code dans sa version tolérante aux pannes (c.-à-d. qui peut se sauver et se restaurer) était réalisée avec une passe de prétraitement (ou *preprocessing*). DOME constitue une des premières tentatives de transformation semi-automatique de code en sa version tolérante aux pannes. En ce qui concerne le partitionnement des données, DOME se limitait à des partitionnements préétablis : réplication des données et distribution par bloc statique [11].

***DRMS* (*Distributed Resource Management System*)** [97] était un environnement pour développer des applications reconfigurables selon le paradigme SPMD. La reconfigurabilité portait sur les structures de données utilisées : des tableaux à N dimensions. DRMS accomplissait la reconfiguration par l'intermédiaire de la réalisation de points de reprise distribués et l'utilisation d'un nouveau modèle de programmation appelé modèle *DRMS* (ou modèle *SOP*) [93].

Le modèle DRMS étendait les modèles SPMD et MPMD avec les concepts *SOQ* (*Schedulable and Observable Quanta*) et *SOP* (*Schedulable and Observable Points*). Un *SOP* marquait la transition d'un *SOQ* à l'autre. Autrement dit, un *SOQ* correspondait à une région du code délimitée par deux *SOPs*. Le calcul parallèle était vu comme une succession de SOQs, et chaque SOQ était structuré avec 4 sections. Ces dernières comportaient (1) la définition des ressources de calcul, (2) la définition des données (cette définition incluait partitionnement et reconfiguration des données), (3) la définition de code de contrôle (dépendant du nombre de processus) et enfin (4) la définition du calcul.

La structuration en SOQs permettait à *DRMS*, dans le cas d'applications SPMD classiques, de sauver séparément les données dites globales (ou communes à tous les processus) des données présentes au sein des SOQs. Les premières étaient sauvées une seule fois car identiques à tous les processus (sauvegarde d'un segment de données). Quant aux secondes elles étaient sauvées pour chaque processus. Ainsi, DRMS exploitait en détail le paradigme SPMD pour réduire la taille des points de reprise.

Pour bénéficier de la reconfigurabilité, le programmeur devait utiliser les tableaux à N dimensions fournis par DRMS et se conformer au modèle de programmation de DRMS. Une implantation de DRMS existait pour les machines `IBM SP2` et était limitée aux programmes écrits en FORTRAN 90.

Cactus [30, 2, 58] est un framework qui permet le développement indépendant de modules appelés *épines* (ou *thorns* en anglais) et destinés à être assemblés pour accomplir différentes tâches d'une large simulation. Il existe des épines d'infrastructure et des épines d'applications. Les épines sont coordonnées par un composant central au framework appelé *chair* (ou *flesh* en anglais). *Cactus* vise les simulations multidimensionnelles à large-échelle.

La parallélisation est apportée par une épine d'infrastructure spéciale appelée *pilote* (ou *driver* en anglais). Le pilote nommé `PUGH` permet de paralléliser par décomposition régulière de domaine en utilisant une grille de calcul représentant le domaine [149, 148]. Par défaut `PUGH` partage équitablement la grille de calcul entre les processus. Un partitionnement manuel statique est également possible. `PUGH` place également une zone de chevauchement sur chaque processeur.

11. Un distribution par bloc dynamique est fournie pour effectuer de l'équilibrage de charge à l'exécution.

Cette zone correspond à une réplication de frontières et est appelée *zone phantôme* (ou *ghostzone*). Les zones aux frontières sont échangées à chaque pas de simulation par des communications inter-processus utilisant MPI.

Cactus fournit des fonctionnalités avancées telles que le choix entre plusieurs méthodes d'E/S (*FlexIO*, *HDF5*, etc.) ainsi que des fonctionnalités de tolérance aux pannes. L'écriture du point de reprise distribué peut s'effectuer dans un seul fichier ou plusieurs fichiers. La reprise à partir d'un fichier unique de point de reprise distribué est possible sur une topologie différente de celle ayant subi la panne. *Cactus* est très portable et ainsi permet la migration entre machines d'architectures différentes.

3.8.3 Réalisations (semi-)automatiques

La piste de la tolérance aux pannes assistée par les compilateurs n'est pas neuve. En 1995, DOME [5] incluait un outil de transformation qui à partir d'un emplacement de réalisation de point de reprise placé par le programmeur génère le code capable de se restaurer. En 1998, [147] utilise le compilateur pour pouvoir générer des points de reprise portables : fonctionne avec un sous-ensemble du langage C. Dans cette section nous nous intéressons à C^3 et $CPPC$. Ces systèmes généralisent l'approche utilisée par DOME dans lequel les transformations sont limitées aux seuls codes écrits avec DOME. C^3 et $CPPC$ s'appliquent à des codes écrits en langage C (ou FORTRAN). La mise en œuvre de ces systèmes est facilitée par celle de compilateurs source-à-source comme ROSE [135] et Cetus [82].

C³ (Cornell Checkpoint Compiler) [19] est un système récent conçu et développé par une équipe de chercheurs à l'université de Cornell : C^3 transforme des applications MPI écrites en langage C (ou FORTRAN) en des applications capables de sauver et de restaurer leur état automatiquement sur toute machine où le système C^3 est disponible.

C^3 se fonde sur un SPRS (semi-)automatique et de niveau applicatif [86] ainsi que sur le protocole de coordination *RROMP* [18]. Le programmeur place un élément de marquage (*e.g.* : une fonction `PotentialCheckpoint` ou une directive de compilation `CHECKPOINT_HERE`) aux endroits où il souhaite que les points de reprise soient réalisés. Ce marquage est utilisé par le SPRS pour transformer l'application de manière à ce qu'elle puisse sauver et restaurer son contexte aux endroits marqués par le programmeur.

La cohérence entre les points de reprise est assurée par le protocole *RROMP*. Il s'agit d'un protocole de coordination non-bloquant pour la réalisation de points de reprise, et qui est inspiré du protocole *distributed snapshots* de Chandy-Lamport (cf. Section 3.4.4, p. 37). Cependant, RROMP se différencie par son niveau d'implantation : il est implanté au niveau de la spécification MPI, alors que *distributed snapshots* est prévu pour être implanté à un niveau plus bas (*e.g.* : socket, etc.). Ainsi, RROMP est implanté au niveau des primitives de communication MPI.

En pratique, C^3 fournit des fonctions d'emballages (ou *wrapper functions*) pour chaque primitive MPI : l'implantation de RROMP s'effectue au sein de ces fonctions d'emballage [12]. Cette approche permet de conserver une indépendance vis-à-vis de l'implantation MPI utilisée mais n'est pas suffisante pour traiter le cas des opérations collectives (*e.g.* : `MPI_Bcast`, etc.) [23]. Ces dernières sont réécrites sous forme de primitives non collectives. Le surcoût sur les temps d'exécutions qui découle de la réécriture est faible sauf dans certains cas.

C^3 utilise une technique similaire à celle de CoCheck pour l'intégration du protocole de coordination. C^3 réussit une indépendance totale vis-à-vis de l'implantation MPI. Un autre

12. Approche similaire à celle de CoCheck

avantage notable du protocole est qu'il supporte la présence de messages sur le réseau pendant la réalisation de points de reprise. Pour un programmeur, cela signifie plus de flexibilité sur l'endroit de placement des points de reprise dans le code source de son application. En effet, il ne doit pas s'assurer de l'absence de messages sur le réseau comme dans la plupart des autres solutions de niveau applicatif.

Malgré la présence de cette flexibilité, les points de reprise devront être réalisés au même endroit dans le code par tous les processus. Si tel n'est pas le cas pour certains processus, ils réaliseront le point de reprise au prochain emplacement de réalisation de points de reprise qu'ils rencontreront. D'une part, cette situation présente l'inconvénient de retarder d'un temps plus ou moins long [13] la terminaison de la phase de réalisation du point de reprise distribué. D'autre part, elle est susceptible d'occasioner un surcoût à l'exécution de l'application plus important car potentiellement, davantage de messages devront être enregistrés. Enfin, une reprise à partir du point de reprise distribué résultant, souffrira de cette désynchronisation introduite lors de la réalisation du point de reprise.

MERLIN [90] est un outil similaire à C^3. La similarité repose sur l'approche et le protocole de coordination utilisés. Cependant, MERLIN s'intéresse à la tolérance aux pannes d'applications FORTRAN/MPI et, de surcroît, dans des environnements hétérogènes (comme celui des grilles de calculs). Ainsi, MERLIN utilise un format de fichier de point de reprise universel inspiré de Porch [119].

CPPC (Controller/Precompiler for Portable checkpointing) [125, 124] est un framework permettant de sauver (sauvegarder) des applications parallèles utilisant MPI. L'accent est mis sur la portabilité : aussi bien celle des points de reprise réalisés que celles des opérations de CPPC. Pour cela, CPPC fournit un ensemble de directives de compilation (*i.e.* : sous forme de `#pragma`) que le développeur de l'application utilise pour annoter son application. Le développeur utilise les annotations pour marquer les points de sauvegarde (endroits dans l'application où il souhaite déclencher des sauvegardes) et pour indiquer les données à sauver. Des annotations existent également pour indiquer des ensembles d'instructions qui doivent impérativement être ré-exécutés à la reprise : il s'agit principalement d'instructions créant un contexte qui n'est pas sauvé par CPPC (*e.g.* : communicateurs MPI et autres objets dits opaques de MPI). La ré-exécution permet de reconstruire un tel contexte L'ensemble de ces annotations structure l'application en sections : les sections destinées à être ré-exécutées et les autres. Cette approche permet de réaliser des points de reprise séquentiels. Pour assurer la cohérence des sauvegardes, les points de sauvegarde doivent impérativement être placés à des *endroits sûrs*. Il s'agit encore une fois d'endroits où l'absence de messages en transit est garantie.

Une particularité de CPPC par rapport à d'autres solutions de réalisation de points de reprise au niveau applicatif réside dans la manière de définir la période de réalisation des points de reprise. Habituellement, cette dernière est fonction du nombre d'itérations de la boucle de calcul principale. Dans CPPC, elle est fonction du nombre d'appels à la fonction de réalisation de points de reprise. Les deux manières de procéder sont équivalentes pour une boucle munie d'un seul emplacement de réalisation de points de reprise. En présence de multiples emplacements (de points de reprise), l'approche de CPPC évite de réaliser des points de reprise à chaque emplacement rencontré. Un tel comportement est plus souhaitable (ou raisonnable) que de réaliser un point de reprise à tous les emplacements rencontrés au sein de la boucle. Dans tous les cas,

13. Ce temps correspond au temps écoulé entre les deux emplacements de réalisation de points de reprise.

l'espacement temporel entre deux réalisations de points de reprise est difficilement contrôlable par le programmeur.

Récemment, CPPC a été amélioré [126] en automatisant un certain nombre de tâches. À présent, il est en mesure d'automatiquement (1) placer des points de reprise, (2) détecter les instructions qui doivent être ré-exécutées à la reprise et (3) choisir les données à sauver pour chaque point de reprise spécifié.

L'approche (semi-)automatique proposée par C^3 et $CPPC$ semble bien fonctionner pour des applications qui utilisent un langage comme le C ou le FORTRAN. Comparé à ces langages, le C++ dispose d'une sémantique beaucoup plus complexe qui rend très difficile le même type d'approche. Or, C++ est un langage qui devient de plus en plus présent dans le développement logiciel : aussi bien dans le milieu HPC que financier.

3.9 Alternatives et évolutions récentes des solutions de tolérance aux pannes

Face à des pannes qui deviennent de plus en plus fréquentes, notamment à cause d'une augmentation du nombre d'unités de calcul, des optimisations et de nouvelles approches sont envisagées pour améliorer la tolérance aux pannes. Cette amélioration concerne la réduction du coût de la tolérance aux pannes.

Nous commençons par énoncer quelques techniques d'optimisation de la réalisation de points de reprise qui existent déjà et qui ont été mis en œuvre dans certains environnements décrits dans les sections précédentes. Nous nous intéressons ensuite à de nouvelles approches de tolérance aux pannes qui sont explorées actuellement.

3.9.1 Techniques traditionnelles d'optimisation

La taille des points de reprise est un facteur important dans la réalisation des points de reprise car elle conditionne le temps que l'application passera en dehors des calculs. Les techniques que nous présentons ci-après sont issues de la réalisation de points de reprise séquentiels et visent à réduire la taille des données écrites ou à faire en sorte qu'elle ralentisse le moins possible l'application dans son exécution. Il s'agit de techniques complémentaires pour optimiser la réalisation de points de reprise.

Réalisation asynchrone de points de reprise : cette technique consiste à copier sur support de stockage le point de reprise de manière concurrente à l'exécution de l'application [113]. Cela peut se réaliser en bloquant le processus applicatif le temps d'effectuer en mémoire une copie des données à sauver. Ces données sont ensuite écrites pendant que le processus applicatif continue son exécution.

Une alternative simple, qui évite cette recopie mémoire, consiste à utiliser les appels systèmes `fork` (ou similaires). Un tel appel résulte en la création d'un nouveau processus qui est une copie conforme du premier. Il suffit ensuite que ce dernier écrive ses données sur disque pendant que le processus initial continue son exécution. Un avantage d'utiliser ce type d'appel est que, dans des systèmes récents, la duplication du processus s'effectue avec une stratégie *copy-on-write* (COW). En d'autres termes, seul un minimum d'éléments (pages mémoires) sont recopiés initialement. Un élément qui n'est pas initialement recopié, demeure partagé entre les processus jusqu'à ce qu'un processus demande à le modifier. Ce n'est qu'à ce moment qu'une recopie de l'élément est réalisée. Ainsi, avec la stratégie COW, les éléments sont recopiés au fur et à mesure des besoins.

De cette manière la quantité de mémoire nécessaire est moindre. Les différentes mises en œuvres décrites s'appliquent aussi bien pour les points de reprise au niveau applicatif que système même si historiquement cela fut appliqué au niveau système.

Compression des points de reprise : cette technique consiste simplement à compresser les données du point de reprise avant de l'écrire sur disque. Cependant, [115] montre que cette technique n'est intéressante que lorsque le facteur de compression est supérieur au rapport entre la vitesse d'écriture sur le media de stockage et la vitesse de compression. La technique de compression doit donc être rapide – nécessiter peu de temps – et efficace – aboutir à des réductions en taille. Ceci est étroitement lié à la vitesse du processeur, à l'adaptation de l'algorithme aux données cible et à la nature des données.

Points de reprise incrémentaux : cette technique consiste à réaliser des points de reprise de manière incrémentale. Il s'agit de n'écrire d'un point de reprise à l'autre que les différences. La mise en œuvre de cette technique est réalisée principalement au niveau système en utilisant différents mécanismes de gestion de la mémoire. Les algorithmes possèdent des granularités différentes au niveau de la page ou du mot mémoire [116, 88].

Les points de reprise incrémentaux sont intéressants lorsque peu de données sont modifiées d'un point de reprise à l'autre. Dans le cas contraire, le gain obtenu grâce à la réduction de la taille des points de reprise tend à disparaître, et le surcoût introduit par la réalisation des points de reprise peut devenir très important. Par ailleurs, la reprise sur panne à partir d'un point de reprise incrémental peut être plus longue que celle à partir d'un point de reprise simple. En effet, un point de reprise incrémental ne contient que les différences par rapport au point de reprise incrémental précédent. Donc, restituer l'état d'un processus à partir d'un point de reprise incrémental réalisé au temps t nécessite d'une part la lecture en mémoire de tous les fichiers de points de reprise incrémentaux précédents ; et d'autre part, cela nécessite l'application de l'algorithme qui permette de reconstituer l'état du processus au temps t.

Récemment une approche au niveau applicatif et utilisant un compilateur source-à-source a été proposée dans [20, 21].

3.9.2 Réalisation coopérative de points de reprise

Une autre approche consiste à essayer de réduire le nombre de points de reprise réalisés. On essaie de les réaliser de manière plus futée en tenant compte d'informations obtenues de l'environnement d'exécution.

Certaines études sont arrivées à la conclusion que la réalisation de points de reprise de manière périodique n'est pas optimale. Un premier système proposé est le *cooperative checkpointing* [106, 105, 104]. Ce dernier considère des points de reprise déclenchés de manière interne à l'application (*i.e.* : existence d'une instruction de réalisation d'un point de reprise dans le code source de l'application). Ces instructions sont placées par le programmeur qui connaît les endroits intéressants pour placer de telles instructions : emplacements avec une faible empreinte mémoire, emplacements après un calcul très lourd, etc. Cependant, le programmeur ne connaît pas le moment le plus approprié pour déclencher de tels points de reprise. Cette tâche est confiée à une entité de l'environnement d'exécution appelée le *gatekeeper* (ou gardien). Le gardien dispose d'informations relatives à la santé de tout le système ainsi que des informations de prévision de futures pannes. À chaque fois que l'application peut réaliser un point de reprise, elle s'addresse au gardien pour obtenir l'autorisation de réaliser un point de reprise. Ce dernier se fonde sur les informations dont il dispose pour accorder à l'application le droit ou non de réaliser un point de

reprise. Il n'existe pas de réalisation concrète d'un tel système. Seules des simulations sur des traces réelles de fichiers d'enregistrement ont été réalisées. Cette approche peut facilement être adaptée pour des solutions de tolérance aux pannes de niveau système. La requête de sauvegarde issue de l'application est simplement remplacée par une requête de l'environnement vers le SPRD.

Dans tous les cas, la réalisation d'un point de reprise de l'application doit être effectuée suffisamment à l'avance afin que l'application soit sauvée avant l'occurrence de la panne. Il est donc nécessaire d'avoir une bonne estimation du temps nécessaire à la sauvegarde, mais également de disposer d'un système précis de prévision des pannes. Un système imprécis conduirait à la réalisation de points de reprise inutiles. C'est sur un tel système que repose la tolérance aux pannes dite *proactive*.

3.9.3 Réalisation proactive de points de reprise

Jusqu'à présent la tolérance aux pannes et notamment la réalisation des points de reprise étaient envisagés comme de mécanismes réactifs. À savoir, le point de reprise est utilisé pour reprendre l'exécution de l'application après que la panne survienne. Souvent dans les solutions actuelles cela implique un arrêt puis une reprise de l'ensemble des processus de l'application. Ce type de manœuvre peut être très coûteux en présence de taux de pannes élevés tels que ceux qui sont attendus pour les machines pétaflopiques et exaflopiques.

L'approche proactive consiste à prévoir l'occurrence de pannes sur les nœuds de manière à migrer les processus concernés de l'application sur des nœuds sains. Bien entendu, la migration doit être accomplie avant que la panne ne survienne. Ainsi, la tolérance aux pannes réactive répond à l'occurrence de pannes tandis que la tolérance aux pannes proactive anticipe leur occurrence.

Le schéma adopté repose sur une interaction entre le gestionnaire de ressources de l'environnement d'exécution et le logiciel de surveillance des nœuds [46]. Le gestionnaire de ressources alloue des nœuds à une application. Il reçoit ensuite un *feedback* sur la santé des nœuds et de l'application qui s'y exécute. Les informations communiquées par le système de surveillance sont utilisées pour effectuer d'éventuelles migrations (voire reconfigurations) de l'application. [46] décrit de manière plus détaillée le schéma que nous venons de présenter et propose une classification de différentes options envisageables lors de sa mise en œuvre.

3.9.4 Vers des environnements coopératifs

Récemment est apparu une infrastructure nommée CIFTS (*Coordinated Infrastructure for Fault-Tolerant Systems*) [61] dont le but est de permettre l'échange d'informations relatives à la tolérance aux pannes et ce, de manière à obtenir un environnement plus résistant aux pannes. Le FTB (*Fault-Tolerant Backplane*) est une pièce centrale à CIFTS : il s'agit d'un framework à publication et abonnement (ou *publish/subscribe*). Les informations échangées sont appelées des *événements de fautes* (ou *fault events*). Des clients s'abonnent à FTB pour être avertis d'événements de fautes mais également pour en publier. La nature des clients est très variée : système d'exploitation, système de fichiers, plannificateur de ressources, SPRD, bibliothèques de passage par messages etc. FTB prévoit un certain nombre d'événements dont la signification aura été préalablement acceptée par la communauté de CIFTS. D'autres événements pourront être définis mais ils seront spécifiques au client qui les produit. Par exemple, Open MPI définit l'événement `MPI_NODE_DEAD` pour informer de l'impossibilité de contacter un processus. La connaissance de cet évènement peut permettre aux autres processus de la même application MPI de prendre une

mesure appropriée. De la même manière, l'occurrence d'erreurs d'entrées/sorties au niveau d'un système de fichiers (SF1) peut générer un événement spécifique qui permetterait à un gestionnaire de ressources (ayant souscrit à ce type d'événement) d'utiliser un autre système de fichiers (SF2) pour les futures applications. Une infrastructure comme CIFTS a donc pour but de rendre disponible l'information relative aux fautes et d'aboutir à une meilleure résistance de l'ensemble du système.

3.10 Résumé de l'état de l'art

Dans ce chapitre nous avons dressé un état de l'art de la tolérance aux pannes dans les grappes de PCs. Les grappes de PCs sont parmi les architectures les plus populaires et les plus répandues pour subvenir aux besoins en puissance de calcul, mais elles sont de plus en plus sujettes à des pannes. Les différents types de pannes présentes dans les grappes sont exposées mais nous nous concentrons sur les *pannes par arrêt*. D'après la littérature, ce type de pannes semblent être très présentes dans les environnements de grappes et résultent en grande partie du passage à l'échelle de ces environnements et des applications qui s'y exécutent. Au cœur des travaux visant à tolérer les pannes par arrêt, se trouvent les *protocoles de reprise par retour arrière* et la *réalisation de points de reprise*. À cause de l'importance de ces protocoles, nous en avons décrit les principes puis nous en avons dressé une classification qui prend en compte des protocoles de niveau applicatif qui mettent en œuvre une coordination sans communications. Ensuite, chaque classe a été développée plus en détails par l'exposition d'un représentant significatif. Par ailleurs, nous avons dressé une classification des réalisations en tolérance aux pannes selon le niveau de réalisation des points de reprise (système ou applicatif) et pour chaque classe nous avons donné plusieurs exemples de réalisations concrètes. De l'exposition de ces réalisations se dégage une préférence pour les protocoles de points de reprise coordonnés et bloquants d'une part, et d'autre part une préférence pour les solutions de niveau système que nous retrouvons dans la plupart des solutions actuelles de tolérance aux pannes (Open MPI, DMTCP, etc.). Il existe néanmoins des solutions au niveau applicatif (semi-)automatique et fondées sur des frameworks mais qui semblent moins présentes. Enfin, nous avons présenté quelques évolutions récentes qui visent d'une part à permettre la réalisation de points de reprise en se fondant sur la prédiction des pannes, et d'autre part en mettant un place des « écosystèmes » tolérants aux pannes.

Chapitre 4

MoLOToF : un nouveau modèle de tolérance aux pannes

Dans ce chapitre nous présentons *MoLOToF* : un nouveau modèle de programmation pour la tolérance aux fautes à faible coût (ou *Model for Low-Overhead Tolerance of Faults*) qui permet le développement d'applications de calcul distribué avec une tolérance aux pannes efficace.

Le chapitre s'articule en deux parties. Dans la première partie sont exposés les objectifs de MoLOToF en insistant notamment sur les applications visées (Section 4.1.1) et sur les pannes considerées (Section 4.1.2) par le modèle. Dans la deuxième partie sont présentées les deux principaux concepts sur lesquels se fonde MoLOToF : les *squelettes tolérants aux pannes* (Section 4.2.1) et *les collaborations* (Sections 4.2.3 et 4.2.4).

4.1 Objectifs

Les solutions pratiques existant dans la littérature (cf. Section 3.6) proposent une tolérance aux pannes fondée sur la réalisation de points de reprise avec une volonté d'impliquer le moins possible le programmeur. C'est le cas des solutions dites de niveau système. C'est aussi le cas des solutions dites de niveau applicatif qui utilisent des compilateurs source-à-source tels que *Cetus* [82] et *ROSE* [135] pour transformer un programme dans sa version tolérante aux pannes. Cette approche conduit à une séparation nette entre l'application et la tolérance aux pannes et ne favorise pas les intéractions entre les deux couches (ou niveaux). MoLOToF est un nouveau modèle pour le développement d'applications de calcul distribué tolérantes aux pannes. Ce modèle vise à atteindre une **efficacité** et une **portabilité** du *Système de Points de Reprise Distribués* (ou SPRD) qui sont meilleures que celles apportées par les SPRD existants. MoLOToF se situe donc au niveau applicatif et contrairement aux solutions existantes, MoLOToF se fonde sur des collaborations entre le programmeur et le SPRD pour mettre en place une tolérance aux pannes efficace.

4.1.1 Applications parallèles considérées

Les applications visées par MoLOToF sont des applications de calcul distribué (ou parallèle) qui sont exécutées sur des grappes de PCs. Ces applications sont caractérisées par la quantité de calculs mis en jeu et la durée d'exécution qui en résulte. Lorsque cette dernière est suffisamment longue, elle peut compromettre l'exécution car la probabilité qu'une panne survienne augmente avec le temps. Par ailleurs, les applications considerées s'inscrivent dans un contexte plutôt

industriel et de ce fait sont soumises à des contraintes de temps. Ces dernières sont généralement des *contraintes temps réel souples* (par opposition aux *contraintes temps réel strictes*) et par conséquent n'ont pas de conséquences extrêmement coûteuses ni catastrophiques lorsqu'elles ne sont pas respectées de manière stricte [25].

Cela dit, les solutions potentielles de tolérance aux pannes doivent être *efficaces* de manière à respecter au mieux ces contraintes temporelles. En particulier, le ralentissement de l'application en l'absence de pannes doit être raisonnable et, en présence de pannes, le temps de calcul perdu à cause de la panne doit être faible et la reprise doit être rapide.

4.1.2 Pannes considérées

Les pannes envisagées par MoLOToF sont des pannes par arrêt de processus ou de machines. Comme annoncé à la section 3.2.2, il s'agit des pannes qui concernent le plus les applications de calcul distribué dans un contexte de grappes. Ces pannes peuvent être temporaires ou permanentes.

La solution de tolérance aux pannes choisie par MoLOToF repose sur la réalisation de points de reprise distribués au niveau applicatif. Il en découle que MoLOToF ne prend pas en compte les pannes permanentes logicielles. Ces dernières résultent souvent d'erreurs de logique dans l'application et sont supposées être traitées en phase de développement et de débogage de l'application.

Les points de reprise permettent en revanche de faire face à des pannes de processus temporaires mais également à des pannes de machines temporaires et permanentes à condition que les points de reprise restent accessibles. Suite à une panne de machine temporaire, la machine fautive peut être relancée et les données qui y sont sauvées sont de nouveau accessibles. Suite à une panne de machine permanente, la reprise est possible si les points de reprise se trouvent sur un support de stockage stable. Un tel support est une abstraction représentant un stockage parfait (*i.e.* : infaillible) [43] et peut être simulé par différentes stratégies.

Par la suite nous nous concentrons sur les mécanismes centraux au modèle MoLOToF qui concernent la capacité à réaliser des points de reprise distribués au niveau applicatif de manière efficace. Les questions relatives à la détection de pannes et la simulation de supports de stockage stables sont orthogonales aux problèmes traités par MoLOToF. Comme nous le verrons par la suite, elles peuvent être ajoutées dans MoLOToF *a posteriori*.

4.2 Fondations du modèle

Afin de tolérer les pannes décrites à la section 4.1.2 de manière efficace, MoLOToF se fonde sur des points de reprise au niveau applicatif. Les points de reprise constituent une technique générale et très répandue en tolérance aux pannes. Leur mise en place dans un calcul distribué requiert d'une part de pouvoir sauver et restituer l'état de processus individuels ; et d'autre part, d'assurer une cohérence entre les différents points de reprise individuels réalisés.

> **Principe 4.1.** *MoLOToF propose une solution s'appuyant sur une structuration des applications de calcul distribué avec des* squelettes tolérants aux pannes.

Cette approche permet de concentrer la tolérance aux pannes sur les points importants de l'application ; elle contribue à la sauvegarde et à la restitution ; et fournit une interface pour rendre effective la collaboration entre le programmeur et le SPRD.

4.2.1 Squelettes tolérants aux pannes

Principe 4.2. *Un* **squelette tolérant aux pannes** *de MoLOToF correspond à une boucle ou une succession de boucles chacune munie d'un* point de reprise. *Le reste du corps des boucles comporte des* phases de calculs *et éventuellement* des phases de communications.

Définition 4.2.1. *Un squelette tolérant aux pannes* **séquentiel** *(ou* **simple***) comporte uniquement des calculs. Un squelette tolérant aux pannes* **parallèle** *(ou* **étendu***) comporte également des communications.*

Définition 4.2.2. *Une boucle au sein d'un squelette tolérant aux pannes est dite* **boucle tolérante aux pannes** *et son état, constitué de la valeur de l'itérateur de boucle, est systématiquement intégré au point de reprise. La boucle peut être de deux types : une* boucle à nombre d'itérations fixe *(boucle* `for`*) ou une* boucle à nombre d'itérations inconnu *(boucle* `while`*).*

La figure 4.1a illustre schématiquement un squelette séquentiel très simple où `FT_Loop` désigne une boucle tolérante aux pannes. La figure 4.1b illustre un squelette parallèle très simple qui ne diffère que par la présence d'une phase de communication représentée par la fonction `communications`.

```
1 FT_Seq_Skel
2 {
3       FT_Loop
4       {
5             calculations()
6
7             checkpoint()
8       }
9 }
```

(a) Squelette séquentiel.

```
1 FT_Par_Skel
2 {
3       FT_Loop
4       {
5             calculations()
6             communications()
7             checkpoint()
8       }
9 }
```

(b) Squelette parallèle.

FIGURE 4.1 – Squelettes tolérants aux pannes de MoLOToF.

Le choix de l'ordre des opérations dans les squelettes est un choix arbitraire dépendant de l'implantation et laissé à la discrétion du fournisseur des squelettes. Par exemple, les squelettes peuvent être agencés pour contenir plusieurs phases de calculs et de communications au sein de la même boucle de calcul comme à la figure 4.2. Entre les ensembles de phases, il peut exister des opérations optionnelles (cf. ci-dessous) représentées par des points de suspension dans les figures. Un autre agencement possible des squelettes prend la forme d'une succession de boucles de calcul séparées éventuellement par des opérations (cf. Figure 4.3). Les opérations optionnelles auquelles on se réfère doivent être *légères* :

Définition 4.2.3. *Une* **opération légère** *désigne une opération ou un ensemble d'opérations que l'on retrouve en dehors des opérations de calculs : par exemple entre des squelettes ou entre des boucles tolérants aux pannes. Par ailleurs, une* opération légère *est caractérisée par :*

1. *une* **durée d'exécution courte** *(voire très courte) par rapport à celle d'une itération de calcul ; et*

```
1  FT_Seq_Skel
2  {
3        FT_Loop
4        {
5              calculations1()
6
7              checkpoint1()
8
9              // ...
10
11             calculations2()
12
13             checkpoint2()
14
15             // ...
16       }
17 }
```

(a) Squelette séquentiel.

```
1  FT_Par_Skel
2  {
3        FT_Loop
4        {
5              calculations1()
6              communications1()
7              checkpoint1()
8
9              // ...
10
11             calculations2()
12             communications2()
13             checkpoint2()
14
15             // ...
16       }
17 }
```

(b) Squelette parallèle.

FIGURE 4.2 – Variante des squelettes tolérants aux pannes de MoLOToF : boucle de calcul unique avec phases de calculs et de communications multiples.

2. *une* **invariance** *de l'opération par rapport au flot de contrôle de l'application. En d'autres termes, les valeurs des variables manipulées par une opération légère ne sont pas utilisées pour prendre des décisions modifiant le flot de contrôle : ce dernier en est indépendant.*

Les propriétés de *durée courte* et d'*invariance* sont des propriétés qui doivent être respectées pour assurer la *performance* et maintenir la *correction* de l'application lors de la reprise sur panne. Elles découlent du mécanisme de sauvegarde-reprise adopté par MoLOToF (cf. Section 4.2.2).

Une application de calcul distribué développée selon MoLOToF dote chacun de ses P processus d'un squelette parallèle enchaînant des phases de calcul et de communication. Ce squelette constitue la trame d'un processus. Les squelettes définis ne sont pas nécessairement identiques sur tous les processus : ceci dépend notamment du paradigme de programmation parallèle employé.

Dans l'absolu, la trame d'un processus n'est pas limitée à un unique squelette parallèle. Elle peut en comporter d'autres et peut aussi être completée par des squelettes séquentiels comme illustré à la figure 4.4. Cette dernière met en évidence les *relations verticales* et les *relations horizontales* qu'entretiennent les squelettes. Les relations verticales sont établies au sein d'un même processus entre deux squelettes consécutifs $S1$ et $S2$ lorsque le résultat de $S1$ est consommé par $S2$. En particulier, les relations verticales sont unidirectionnelles. Les relations horizontales sont établies entre deux squelettes appartenant à des processus différents. Elles sont bidirectionnelles et matérialisent des communications entre les processus. À noter que des squelettes parallèles ne peuvent communiquer qu'avec des squelettes portant le même identifiant. Par exemple, à la figure 4.4 le squelette `Par_Skel1` ne peut communiquer avec le squelette `Par_Skel2`.

4.2.2 Mécanisme de sauvegarde-reprise sous MoLOToF

MoLOToF vise à fournir de la tolérance aux pannes à des applications de calcul distribué grâce à la réalisation de points de reprise distribués de niveau applicatif. Sauver et restaurer une

```
1  FT_Seq_Skel
2  {
3        FT_Loop1
4        {
5              calculations1()
6
7              checkpoint1()
8        }
9
10       // ...
11
12       FT_Loop2
13       {
14             calculations2()
15
16             checkpoint2()
17       }
18
19       // ...
20 }
```

(a) Squelette séquentiel.

```
1  FT_Par_Skel
2  {
3        FT_Loop1
4        {
5              calculations1()
6              communications1()
7              checkpoint1()
8        }
9
10       // ...
11
12       FT_Loop2
13       {
14             calculations2()
15             communications2()
16             checkpoint2()
17       }
18
19       // ...
20 }
```

(b) Squelette parallèle.

FIGURE 4.3 – Variante des squelettes tolérants aux pannes de MoLOToF : boucles de calculs multiples avec phase de calculs et de communications unique.

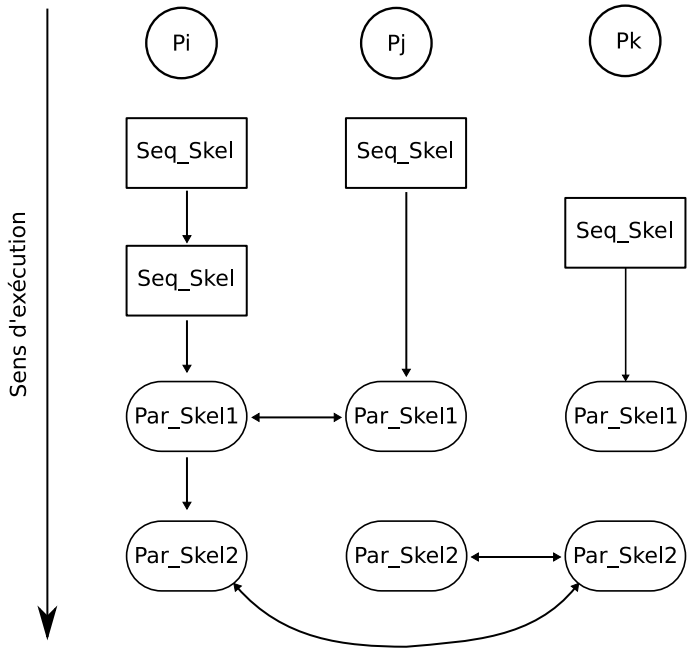


FIGURE 4.4 – Relations possibles entre squelettes.

application revient à être capable de sauver un contexte d'exécution et les données qui y sont associées, mais également de reconstituer ce contexte au moment de la reprise et d'y restituer les valeurs correspondantes.

MoLOToF structure l'application à l'aide de squelettes dits tolérants aux pannes. En somme, il s'agit de squelettes capables de sauver et restituer leur contexte et les données qui y sont rattachées. L'état peut varier d'un squelette à un autre mais comprend au moins l'indice de la boucle et d'autres données propres au squelette dont le type du squelette (séquentiel ou parallèle) ainsi que l'identifiant du squelette qui est unique au sein d'une application. Des exemples d'autres données sont disponibles aux chapitres 5 et 6 où des squelettes sont mis en œuvre respectivement dans le cadre des modèles *Single Program Multiple Data* (SPMD) et *Master-Worker* (MW) (ou Maître-Travailleur).

Principe 4.3. *MoLOToF distingue deux modes d'exécution pour les processus applicatifs : un* mode normal *et un* mode reprise.

Ces deux modes permettent de différencier un processus qui suit son cours normal d'exécution d'un processus participant à une reprise sur panne. En mode normal d'exécution, le processus vérifie simplement si la condition de réalisation de point de reprise est vérifiée avant d'écrire le point de reprise sur stockage stable. En cas de réalisation de point de reprise, il doit veiller à garantir la cohérence de ce dernier par rapport à ceux réalisés sur les autres processus. Cette précaution est nécessaire pour les squelettes parallèles et dépend du protocole de coordination utilisé.

En mode reprise, le processus victime d'une panne est relancé à partir de son point de reprise le plus récent et qui est cohérent avec l'état actuel des processus non défaillants. Bien sûr, il peut arriver que pour maintenir la cohérence, d'autres processus doivent effectuer un retour arrière. Ceci dépend notamment du protocole de reprise par retour arrière mis en œuvre (cf. Section 3.3). La manière de restaurer le contexte d'exécution est détaillée ci-après.

Principe 4.4. *MoLOToF procède par* ré-exécution *pour reconstituer le contexte d'exécution qui est présent au moment de la sauvegarde.*

Deux types de ré-exécution sont envisagés selon qu'elle concerne les squelettes ou non. Pour des contextes autres que ceux des squelettes, la ré-exécution est identique à l'exécution en mode normal. Il s'agit de réinitialiser un certain nombre de variables. Souvent, les variables en question sont en lecture seule ou possèdent une initialisation avec un coût négligeable en temps (cf. opération légère).

La ré-exécution au niveau des squelettes diffère légèrement. À la rencontre d'un squelette pendant la reprise, le système de tolérance aux pannes s'assure qu'il s'agit bien du squelette dont le contexte a été sauvé dans le point de reprise. Si tel n'est pas le cas, le squelette est simplement *traversé* : aucune opération définie au sein du squelette n'est effectuée. S'il s'agit du bon squelette, il est *exécuté en mode reprise.* Dans ce mode, les calculs et les communications sont évités jusqu'à atteindre l'instruction de point de reprise à l'origine du point de reprise. À ce moment, les données sont rechargées à partir du point de reprise et l'exécution reprend son

cours normal. Dans cette étape seules les opérations légères sont ré-exécutées. Ainsi, l'exécution en mode reprise consiste à éviter l'exécution de phases du squelette consommatrices en temps et situées en amont du point de reprise. Cette manière de procéder permet d'atteindre rapidement le point de la sauvegarde et de restituer en même temps le contexte applicatif.

Principe 4.5. *MoLOToF ne considère qu'un niveau de profondeur pour le contexte d'exécution.*

Par exemple, le placement de points de reprise au sein de la fonction de calcul appelée dans le squelette n'est pas supporté. De manière plus générale, MoLOToF ne permet pas de sauver et de restituer des contextes imbriqués. De ce fait, les squelettes ne supportent pas de sauver des calculs situés à l'intérieur de fonctions.

MoLOToF propose donc un contexte unique pour la tolérance aux pannes. Bien que ce principe paraisse contraignant au premier abord, nous soutenons que de nombreuses applications de calcul distribué peuvent s'y conformer. Par ailleurs, ceci permet de conserver un modèle simple pour le programmeur.

Principe 4.6. *Dans MoLOToF, le SPRD prend en charge l'*écriture *et la* lecture *des points de reprise mais également la* cohérence *entre les points de reprise.*

Dans le cadre du mécanisme de sauvegarde-reprise, le SPRD prend en charge l'écriture et la lecture de points de reprise. Ces dernières peuvent être réalisées de différentes manières selon le niveau de portabilité souhaité. Par ailleurs, il met en œuvre un protocole efficace pour assurer la cohérence entre les points de reprise. Enfin, ceci comprend également, en cas de panne, la détermination de la ligne de reprise la plus récente pour reprendre l'exécution de l'application.

Normalement, la conservation de deux points de reprise distribués est suffisante pour faire face à des pannes survenant pendant les phases de tolérance aux pannes (pendant la réalisation du point de reprise ou de la reprise sur panne) [76]. Néanmoins, si l'espace de stockage le permet, conserver plusieurs points de reprise est en soit une forme de mesure à l'encontre des fautes de corruption de données qui peuvent survenir sur le support de stockage s'il n'est pas stable.

Le squelette tolérant aux pannes de MoLOToF est au cœur de diverses collaborations entre d'une part le programmeur et le SPRD et d'autre part entre le SPRD et l'environnement d'exécution. Les deux sections suivantes précisent ces collaborations.

4.2.3 Collaborations entre programmeur et système de tolérance aux pannes

Dans le cadre de la collaboration entre le programmeur et le SPRD, chacun se voit attribuer des responsabilités (ou rôles). Le SPRD est chargé de fournir les squelettes tolérants aux pannes et de guider le programmeur dans leur utilisation directement ou par l'intermédiaire de MoLOToF. Le programmeur est amené à structurer son application avec des squelettes tolérants aux pannes. Ceci constitue une première collaboration dite **collaboration de placement** et dans laquelle :

Principe 4.7. *Le programmeur place les squelettes tolérants aux pannes aux endroits intenses en calcul où l'application passe le plus de temps.*

Ensuite, le squelette tolérant aux pannes ne sauvegarde par défaut (dans un point de reprise) que ses données propres. Ceci comprend au minimum l'itérateur de la boucle tolérante aux pannes. Mais, le calcul défini par le programmeur peut mettre en jeu d'autres données qu'il est nécessaire de sauver pour la correction du point de reprise. Il en résulte une deuxième collaboration dite **collaboration de correction** dans laquelle :

Principe 4.8. *Le programmeur renseigne le squelette sur les données additionnelles à sauver pour assurer la correction du point de reprise.*

Les variables concernées sont des variables dont le contenu est modifié au sein des calculs définis par le programmeur. Ce sont donc des variables en écriture qui correspondent typiquement à des variables de résultats.

Cette manière de procéder favorise l'obtention de points de reprise contenant un minimum de données et se distingue notamment des autres approches existantes qui par défaut intègrent toutes les données. Ainsi, cette stratégie constitue également une **collaboration de performance** car elle conduit à minimiser la taille des points de reprise. Le SPRD doit donc fournir le moyen de sélectionner les données. Dans le contexte de la collaboration de performance, le SPRD peut également fournir au programmeur le moyen de préciser la *manière de sauver des données*. En effet, des données peuvent avoir une représentation compacte qui réduit la taille des points de reprise. Le programmeur peut alors fournir la relation de passage entre les représentations. Une transformation classique est de demander la compression des données. Une autre est de pouvoir reconstruire une donnée à partir de données sauvées.

Enfin, il existe entre le SPRD et le programmeur une troisième collaboration dite **collaboration de fréquence** et dans laquelle :

Principe 4.9. *Le programmeur peut contrôler la fréquence de réalisation des points de reprise.*

Ceci revient à contrôler le nombre de points de reprise qui seront réalisés. Le nombre maximal de points de reprise est limité par le nombre total d'itérations des boucles tolérantes aux pannes. Compte-tenu du temps moyen entre deux pannes (ou Mean Time Between Failures (MTBF)) de la machine cible et la durée des itérations, le programmeur peut affecter une période de réalisation de points de reprise adaptée.

4.2.4 Collaboration entre système de tolérance aux pannes et environnement

Les collaborations décrites dans la section précédente permettent de profiter de la connaissance que possède le programmeur de son application dans le but d'obtenir une tolérance aux pannes efficace. Cependant, elles ne prennent pas en considération des informations relatives à la tolérance aux pannes que peut fournir l'environnement d'exécution. En particulier, ce dernier peut détenir des informations intéressantes quant aux pannes futures et demander aux applications concernées d'entreprendre des actions préventives. De manière certaine, l'environnement a connaissance d'interruptions plannifiées résultant de sessions de maintenance, de l'atteinte de fin de quota de temps (voire de consommation énergétique) alloué à l'application. De manière probabiliste, l'environnement peut faire des prévisions sur les pannes imminentes. L'action à entreprendre est souvent la réalisation d'un point de reprise par l'application concernée et ce, de

façon à minimiser la quantité de temps de calcul. Il peut également s'agir de modifier la fréquence de réalisation de points de reprise. Le SPRD doit donc être capable de réaliser une sauvegarde de l'application suite à la réception d'un événement externe. C'est ce que nous désignons par **fonctionnement piloté**.

> **Principe 4.10.** *MoLOToF rend possible la mise en œuvre d'un* **fonctionnement piloté** *pour une tolérance aux pannes au niveau applicatif.*

4.3 Résumé du modèle MoLOToF

Dans ce chapitre, nous avons présenté MoLOToF : un nouveau modèle qui permet le développement d'applications de calcul distribué tolérantes aux pannes. Pour y parvenir MoLOToF introduit le concept de squelettes tolérants aux pannes. Ces derniers permettent (1) de concentrer la tolérance aux pannes sur les endroits importants de l'application (ceux qui consomment du temps) ; (2) de fournir au programmeur une interface pour pouvoir contrôler et rendre la tolérance plus efficace, et (3) de maintenir le mécanisme de sauvegarde-reprise simple à utiliser.

Dans les chapitres 5 et 6, nous exposons le résultat de la mise en œuvre de MoLOToF. Par rapport au modèle général présenté ici, nous prenons en considération dans le SPRD le paradigme de parallélisation. Plus particulièrement, nous nous intéressons aux paradigmes *Single Program Multiple Data* (SPMD) et *Master-Worker* (MW).

Chapitre 5

FT-GReLoSSS : un nouveau framework SPMD tolérant aux pannes

FT-GReLoSSS (Fault-Tolerant-Globally Relaxed Locally Strict Synchronization SPMD) est un nouveau framework SPMD (*Single Program Multiple Data*) tolérant aux pannes qui résulte de l'application des principes du modèle MoLOToF (cf. Chapitre 4) à une famille d'algorithmes du paradigme de programmation SPMD.

Après une présentation du type d'applications considérées (Section 5.1), nous introduisons l'architecture du framework FT-GReLoSSS (Section 5.2). Cette dernière introduit un certain nombre de concepts relatifs à SPMD et à la tolérance pannes qui sont explicités juste après (Sections 5.3 et 5.4 respectivement). Enfin, nous présentons un protocole original qui permet à FT-GReLoSSS d'avoir un fonctionnement piloté (Section 5.5) et nous donnons différents détails d'implantation de FT-GReLoSSS (Section 5.6).

5.1 Types de programmes SPMD considérés

Dans le paradigme de programmation parallèle SPMD, un programme unique est écrit et il est exécuté par plusieurs processus mais sur des données différentes. Le cœur des applications SPMD qui nous intéressent consistent en une boucle principale dans laquelle s'alternent des phases de calculs et de communications. Chaque itération correspond à une *super-étape* (ou *superstep*), et les algorithmes considérés suivent donc globalement le schéma des algorithmes BSP. Cependant, ils s'en éloignent sur deux points importants (cf. Section 3.4.5, p. 38) :

- Pour gagner en efficacité à l'exécution sur des architectures parallèles (notamment sur des architectures de grande taille), nous considérons des algorithmes SPMD sans synchronisation globale entre les super-étapes. Chaque processus entame sa super-étape suivante dès qu'il a fini ses propres communications : synchronisation relaxée comme celle du modèle PRO [55].
- Pour gagner en efficacité de checkpointing lorsque l'on veut réaliser de la tolérance aux pannes, chaque processus termine sa phase de communication et passe à une nouvelle phase de calcul seulement quand toutes ses communications sont totalement achevées. Il ne peut donc y avoir de recouvrement des communications et des calculs d'un processus.

Pendant les communications, les processus s'échangent des données correspondant à des données initiales du problème ou des résultats intermédiaires. Les communications entre les processus peuvent être très variées. Cependant, dans une majorité d'applications, elles se réalisent selon des schémas prévisibles qui peuvent être spécifiés par le programmeur. C'est le cas des applications

que nous considérons.

Un premier type d'applications SPMD que nous considérons met en jeu un domaine (souvent à 2 ou 3 dimensions) qui modélise le problème à résoudre. Ce domaine est décomposé de manière régulière et fixe entre les processus de l'application. Chacun des processus réalise des calculs sur la partie du domaine (ou sous-domaine) qui lui a été attribuée. Compte-tenu de la dépendance existant entre les éléments situés aux frontières des sous-domaines, les processus effectuent après chaque phase de calculs une phase de communications consistant en des échanges de frontières. Une phase de calculs et de communications constituent une super-étape et elles sont réalisées de manière itérative jusqu'à ce que la convergence vers la solution soit atteinte. Il s'agit donc d'applications à décomposition de domaine régulière et fixe qui mettent en jeu des échanges de frontières fixes. Un exemple de ce type d'applications est celui des relaxations de Jacobi comme celle que nous avons implanté et que nous présentons à la section 5.4.3.

Le second type d'applications SPMD que nous considérons se base toujours sur une décomposition en domaines des données initiales. Mais les calculs d'un processus ne nécessitent pas seulement ses données locales et celles des frontières des processus voisins. Il s'agit d'algorithmes où chaque processus à besoin de lire des données ou des résultats intermédiaires localisés sur de nombreux processus, parfois même sur tous les processus de l'application distribuée. Pour satisfaire ce besoin on utilise fréquemment une *circulation de données* : les données requises par les processus sont propagées depuis chaque processus vers un processus voisin, qui les lit et les retransmet à son tour. Ces données forment une *partition circulante*. Selon les besoins de l'algorithme on utilise une seule partition circulante globale, qui circule sur tous les processus, ou bien des partitions circulantes partielles qui circulent en parallèle sur des sous-ensembles de processus. Dans tous les cas, ces algorithmes utilisent plusieurs structures de données pour représenter les données circulantes et les données purement locales, ainsi qu'un schéma de circulation conçu pour optimiser les communications sans saturer les capacités de communication du réseau d'interconnexion sous-jacent. Un exemple de telles applications est le produit de matrices denses que nous avons implanté et que nous décrivons à la section 5.3.4. Ces dernières mettent en jeu deux domaines correspondant aux matrices à multiplier.

Un troisième type d'applications SPMD que nous considérons met en jeu, comme pour les relaxations de Jacobi, un domaine unique qui est décomposé de manière régulière entre les processus. Cependant, à la différence des relaxations de Jacobi, le domaine décomposé est en réalité un sous-domaine du domaine initial et son étendue est amenée à varier d'une super-étape à l'autre. Ceci résulte en des frontières irrégulières et dynamiques qui doivent être échangées entre les processus. L'application de *Swing Gazier* (cf. Chapitre 2) que nous avons développée en collaboration avec EDF R&D durant notre thèse fait parti de ce type d'applications à décomposition régulière de domaine, variable dans le temps, avec des échanges de frontières irrégulières.

Par la suite, une application de relaxation de Jacobi et une application réalisant un produit de matrices sont utilisées pour illustrer différents aspects relatifs au framework FT-GReLoSSS (cf Sections 5.3.4 et 5.4.3).

5.2 Architecture du framework FT-GReLoSSS

Pour faciliter le développement d'applications SPMD tolérantes aux pannes, FT-GReLoSSS impose au programmeur un modèle de développement structuré qui est fondé sur des squelettes tolérants aux pannes respectant les principes du modèle MoLOToF (cf. Chapitre 4). FT-GReLoSSS fournit des squelettes que le programmeur utilise pour coder son application. Un squelette définit une structure et, dans le cas des squelettes proposés par FT-GReLoSSS, cette

structure définit le *modèle de parallélisation*. Le programmeur rend effective cette structure en manipulant un certain nombre de concepts liés aux squelettes et qui font parti de l'architecture de FT-GReLoSSS.

La figure 5.1 illustre cette architecture centrée autour de squelettes : au centre nous retrouvons un squelette SPMD autour duquel gravitent plusieurs autres concepts. Le programmeur est amené à agir (directement ou indirectement) sur les concepts situés au niveau de la ligne en trait discontinu en utilisant les concepts qui se trouvent en dessous. Les concepts au-dessus de cette ligne lui restent opaques. Le programmeur ne crée donc pas ses squelettes mais se fonde sur les squelettes fournis par FT-GReLoSSS pour coder son application.

Les concepts non liés à la tolérance aux pannes font l'objet de la section 5.3 tandis que ceux qui ont trait à la tolérance aux pannes font l'objet de la section 5.4.

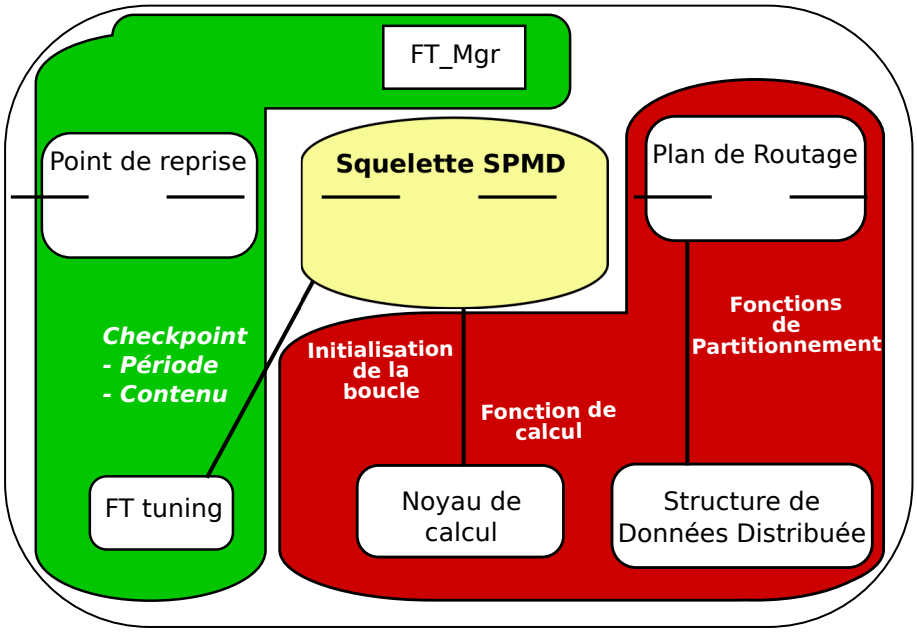


FIGURE 5.1 – Architecture du framework FT-GReLoSSS.

5.3 Concepts SPMD du framework FT-GReLoSSS

5.3.1 Squelettes proposés par FT-GReLoSSS

FT-GReLoSSS propose deux variations de squelettes. La première correspond au cas où le nombre de super-étapes est connu à l'exécution. La seconde correspond au cas où le nombre de super-étapes n'est pas connu en avance. Dans ce cas, l'arrêt peut dépendre des données initiales ainsi que des résultats disponibles au moment de l'évaluation de la condition d'arrêt. Ce type de condition d'arrêt est commun dans les solveurs itératifs où la solution recherchée est approchée par itérations successives. L'arrêt du calcul est effectif lorsque la précision de la solution trouvée est satisfaisante (ou avec un taux d'erreur inférieur à ϵ). Sous certaines conditions, la convergence de la méthode itérative n'est pas garantie. C'est la raison pour laquelle le programmeur peut imposer un nombre maximal d'itérations à l'issue desquelles l'application s'arrêtera même si elle n'a pas convergé vers une (ou la) solution. Il est donc intéressant de pouvoir combiner la condition d'arrêt à nombre de super-étapes fixe avec la condition d'arrêt à nombre de super-

étapes inconnu. La deuxième variation de squelette proposée par FT-GReLoSSS permet ce type de condition d'arrêt.

Une version simplifiée de ces squelettes est donnée aux figures 5.2 et 5.3 respectivement. Mis à part le type de la boucle, le cœur de boucle est identique : il se compose d'une phase de calculs (cf. fonction `compute`) suivie d'une phase de communications (cf. fonction `comms`).

```
1  class GReLoSSS_Skel_for
2  {
3       Skel_for_iter sfi;  // Itérateur du squelette
4       int it;
5
6       Domain  *V1, *V2;   // Structure de données double
7                           // (deux tableaux à N dimensions)
8       void execute()
9       {
10           // Init. du plan de routage
11           Routing_plan *rp = new Routing_plan(/*...*/);
12           for (it = sfi.beg(); it != sfi.end(); it = sfi.next())
13           {
14                compute(sfi);     // Phase de calculs
15                rp->comms(sfi);   // Phase de communications
16                swap();           // Echange des tableaux à N dimensions
17           }
18      }
19 };
```

FIGURE 5.2 – Squelettes SPMD fournis par FT-GReLoSSS : Squelette à nombre d'étapes connu.

Comparés au squelette parallèle présent à la figure 4.1b (p. 59) les squelettes de FT-GReLoSSS incluent une phase supplémentaire qui est matérialisée par la fonction `swap` à la fin de la boucle de calcul. Cette phase, ainsi que la structure des squelettes proposés résulte *du modèle de parallélisation* utilisé par FT-GReLoSSS.

5.3.2 Modèle de parallélisation de FT-GReLoSSS

FT-GReLoSSS propose une parallélisation fondée sur l'utilisation d'une double structure de données distribuée : il s'agit de deux tableaux à N dimensions. Ce modèle de parallélisation est illustré à la figure 5.4. La valeur de N est fonction de l'application. Dans ce mode de parallélisation, un tableau (*NDArray 1*) est utilisé pour fournir les données de calcul. Le second tableau (*NDArray 2*) est destiné à contenir les données en entrée pour le calcul de la prochaine super-étape. Ces tableaux sont échangés à la fin de chaque super-étape en vue de la prochaine super-étape. Ainsi, au cours d'une super-étape, le premier tableau est toujours en lecture pendant la phase de calcul et le second tableau est toujours en écriture pendant la phase de communication.

Ce modèle de parallélisation, qui peut paraître contraignant au premier abord, permet d'implanter aussi bien des applications mettant en jeu de la circulation des données initiales (comme dans certaines applications d'algèbre linéaire) que des applications par échanges de frontières (comme dans certaines applications de relaxation parallélisées par décomposition de domaine).

```
1  class GReLoSSS_Skel_while
2  {
3       Skel_for_iter sfi;     // Itérateur 'for' du squelette
4       Skel_while_iter swi;   // Itérateur 'while' du squelette
5       int it1;
6       bool it2;
7
8       Domain  *V1, *V2; // Structure de données double
9                         // (deux tableaux à N dimensions)
10      void execute()
11      {
12           // Init. du plan de routage
13           Routing_plan *rp = new Routing_plan(/*...*/);
14           it1 = sfi.beg();
15           it2 = swi.beg();
16           while ((it1 != sfi.end()) && (it2 != swi.end()))
17           {
18                compute(sfi);     // Phase de calculs
19                rp->comms(sfi);   // Phase de communications
20                swap();           // Echange des tableaux à N dimensions
21                it1 = sfi.next();
22                it2 = swi.next();
23           }
24      }
25 };
```

FIGURE 5.3 – Squelettes SPMD fournis par FT-GReLoSSS : Squelette à nombre d'étapes inconnu.

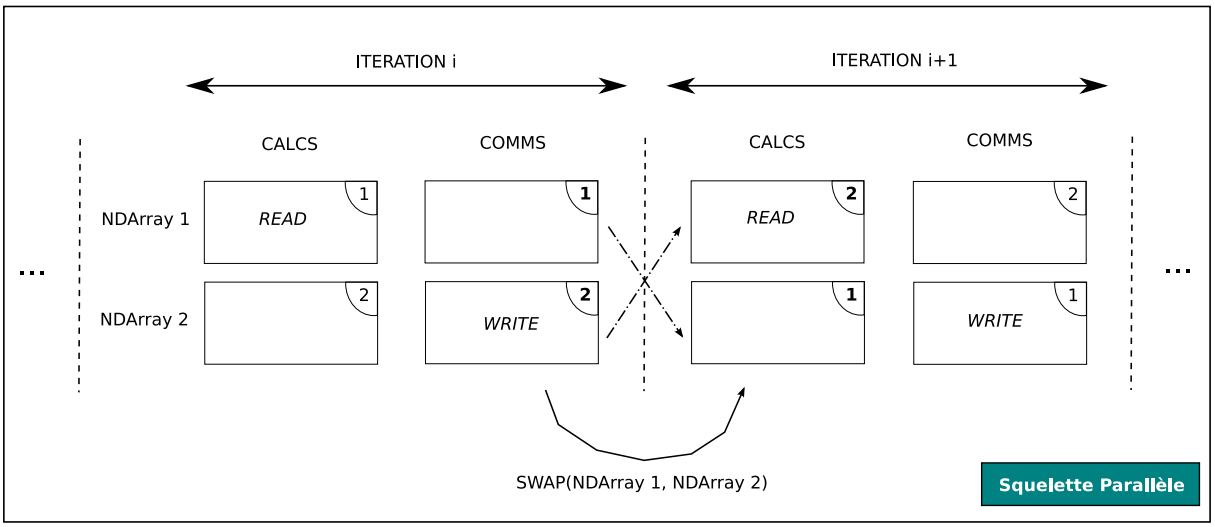


FIGURE 5.4 – Modèle de parallélisation de FT-GReLoSSS.

Le point fort étant que toutes ces applications utilisant FT-GReLoSSS bénéficient quasi-automatiquement d'une tolérance aux pannes efficace.

5.3.3 Noyau de calcul, plan de routage et structure de données distribuée

Les concepts de **noyau de calcul** et de **plan de routage** présents à la figure 5.1 permettent de définir respectivement la phase de calcul et la phase de communication intervenant dans les squelettes de FT-GReLoSSS et sont liés par la **structure de données distribuée** :

- le **noyau de calcul** est un point d'entrée du squelette qui permet d'initialiser la boucle de calcul et de renseigner la fonction de calcul ;
- le **plan de routage** établit, quant à lui, l'ensemble des communications à réaliser pendant la phase de communications de chaque super-étape. Ce plan contient, pour chaque processus applicatif, les données qu'il s'attend à recevoir de la part des autres processus et les données qu'il doit envoyer à tous les autres processus. Le plan de routage est un concept propre à FT-GReLoSSS. Un de ses intérêts est de permettre et faciliter la mise en place de schémas de communication complexes. En particulier, pour des applications large échelle il permet une plannification des communications qui évite de saturer soudainement le réseau. Ce concept trouve son origine dans des travaux de distribution réalisés au cours du travail de thèse et dans lesquels il a démontré son intérêt [28, 128] ;
- la **structure de données distribuée** est un concept classique en programmation parallèle. Dans FT-GReLoSSS, la structure de donnée distribuée correspond à des tableaux à N dimensions. FT-GReLoSSS fournit une interface à de tels tableaux, laissant ainsi au programmeur une liberté quant au choix de l'implantation de la structure de données. La structure de données distribuée occupe une place importante dans FT-GReLoSSS car c'est dans cette structure que le programmeur est amené à définir la manière dont sont distribuées les données entre les processus de l'application. C'est également dans cette structure que sont définies implicitement les communications. Ces deux rôles sont réalisés par la définition de deux *fonctions de partitionnement* qui fournissent des informations quant aux *données possédées* et aux *données requises* à chaque étape du calcul distribué.

La fonction `swap` présente dans les deux squelettes doit être fournie par le programmeur qui peut donc l'implanter par recopie ou par échange de pointeurs. En ce qui concerne la détection de la convergence qui est présente dans le squelette à nombre d'étapes inconnu, elle peut prendre des formes très diverses d'une application à une autre. C'est la raison pour laquelle FT-GReLoSSS laisse la définition du test de convergence entièrement au programmeur.

L'assemblage du noyau de calcul, du plan de routage et de la structure de donnée distribuée permet de mettre sur pied la structure des squelettes SPMD de FT-GReLoSSS. Les squelettes qui en résultent sont suffisamment flexibles pour supporter les trois types d'applications annoncés à la section 5.1.

5.3.4 *Matmult* : Un exemple de produit de matrices denses en parallèle

Dans cette section, nous nous intéressons à la mise en œuvre de *Matmult*, une application réalisant un produit de matrices en parallèle en utilisant FT-GReLoSSS. Nous présentons d'abord l'algorithme de produit de matrices utilisé et montrons comment il s'adapte au modèle de parallélisation de FT-GReLoSSS. Finalement, nous décrivons les différentes étapes de travail du programmeur dans l'écriture de l'application.

5.3.4.1 Description de l'algorithme

Matmult est une application qui réalise un produit de matrices carrées denses A et B de taille N. Le produit s'effectue en parallèle sur un anneau de P processus. Étant donné A et B les matrices opérandes du produit et C la matrice résultat, *Matmult* réalise $C = A \times B$. Tel que c'est illustré à la figure 5.5, la distribution des matrices A et B est fixe sur P processus : initialement chaque processus se voit confier un bloc de N/P lignes de A et un bloc de N/P colonnes de B. La fin du calcul local marque la fin d'une super-étape. À l'issue de cette super-étape, chaque processus envoie son bloc de N/P lignes de la matrice A à son voisin de gauche et reçoit le bloc de N/P lignes de A de son voisin de droite. Après N super-étapes, chaque processus possède un bloc de N/P colonnes de la matrice résultat C.

Dans la terminologie de FT-GReLoSSS, les données possédées par un processus correspondent au bloc de N/P lignes détenues par ce processus. Les données nécessaires pour la prochaine étape de calcul correspondent au bloc de N/P lignes détenues par son voisin de droite.

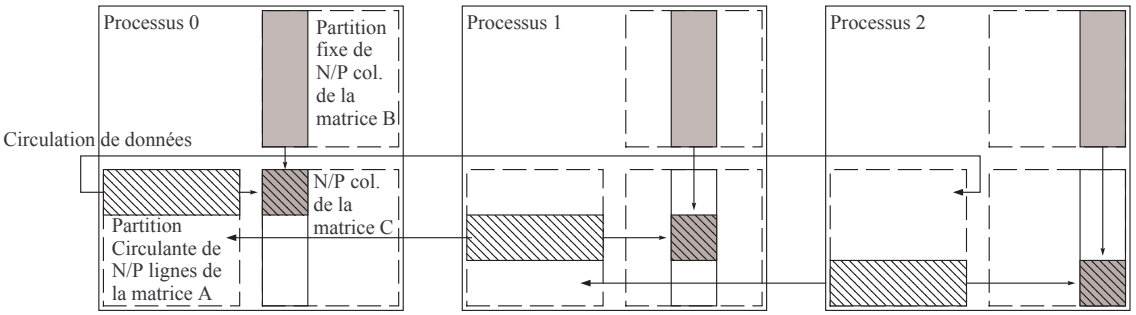


FIGURE 5.5 – Produit de matrice denses en parallèle sur un anneau de processus.

5.3.4.2 Mise en œuvre avec FT-GReLoSSS

Cet algorithme est adapté au modèle de parallélisation proposé par FT-GReLoSSS. La figure 5.6 illustre sa mise en œuvre avec FT-GReLoSSS. À chaque itération de la boucle de calcul (ou super-étape), chaque processus calcule le produit entre les blocs de matrices A et B qu'il possède et écrit le résultat dans le bloc résultat. Sur la figure 5.4, *NDArray 1* correspond au bloc de la matrice A. Pendant la phase de communication, le bloc de la matrice A est lu pour être envoyé au processus voisin de gauche. Le bloc *shadow*, qui correspond au *NDArray 2* à la figure 5.4, est écrit avec la réception du bloc de matrice A provenant du processus voisin de droite.

5.3.4.3 Étapes de développement dans FT-GReLoSSS

Après s'être assuré de la compatibilité de son application avec le schéma algorithmique parallèle proposé par FT-GReLoSSS (cf. Section 5.3.2), la tâche du programmeur se résume aux étapes suivantes :

1. **Interfaçage de son tableau de données avec FT-GReLoSSS** (cf. Figure 5.7). Cette étape consiste à hériter de la classe `Domain` et à implanter un certain nombre de méthodes. La classe `Domain` constitue l'interface aux tableaux à N dimensions proposée par FT-GReLoSSS. Dans ce code, le programmeur utilise des tableaux de la bibliothèque Blitz [13] (l. 5). Parmi les méthodes à définir figurent les deux méthodes de partitionnement

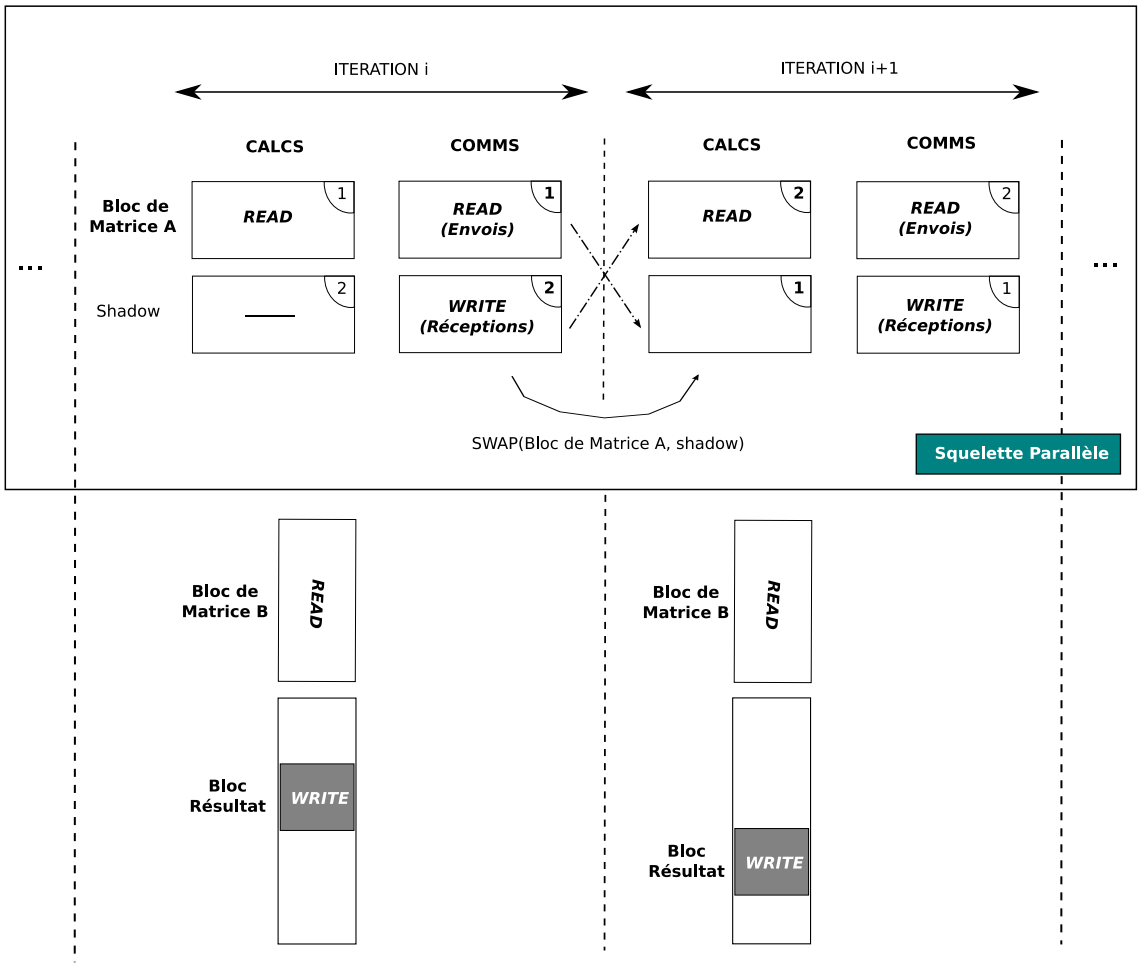


FIGURE 5.6 – Adaptation du "produit de matrice denses en parallèle sur un anneau de processus" au modèle de parallélisation de FT-GReLoSSS.

`data_needed` (l. 16 à 33) et `data_possessed` (l. 35 à 38). Le partitionnement doit être exprimé en termes de `Domain_desc`. Il s'agit d'une classe fournie par FT-GReLoSSS qui constitue une description d'une partie de `Domain` en utilisant un liste d'intervalles (un par dimension). Le programmeur doit définir également des méthodes pour accéder aux éléments du tableau étant donné que l'accès aux éléments peut varier d'une implantation à l'autre (cf. méthodes `lget` et `lset` (l. 40 à 48)). Enfin, le programmeur doit définir la méthode `swap` (l. 50 à 53) qui permet d'échanger le contenu pointé par deux `Domain`. Cette méthode devrait disparaître dans la prochaine version du framework.

2. **Écriture du noyau de calcul** (cf. Figure 5.8). Cette étape revient à hériter de la classe `FT_Calc_Kernel` et à définir la méthode de calcul `compute` ainsi que l'itérateur de la boucle de calcul du squelette.
3. **Écriture de la fonction principale** (`main`) (cf. Figure 5.9). Durant cette étape, le programmeur doit veiller à initialiser FT-GReLoSSS à l'aide d'un appel à une méthode `FT_Mgr::init` et ne pas oublier de quitter proprement en faisant un appel à une méthode `FT_Mgr::finalize` (resp. l. 10 et l. 27). Étant donné que FT-GReLoSSS repose en parti sur MPI, l'appel à `FT_Mgr::init` doit être placé après l'appel à `MPI_Init`. L'appel à `FT_Mgr::finalize` peut être avant ou après `MPI_Finalize`. Entre les appels à `FT_Mgr::init` et à `FT_Mgr::finalize`, le programmeur procède dans l'ordre à l'instanciation du noyau de calcul (l. 12 à 15) qu'il a défini, puis à celle du squelette (l. 17 à 21). Enfin, le programmeur peut lancer l'exécution du squelette (l. 24).

5.4 Concepts de tolérance aux pannes du framework FT-GReLoSSS

Outre les concepts SPMD qui ont été présentés à la section 5.3, l'architecture de FT-GReLoSSS représentée à la figure 5.1 comporte également des concepts liés à la tolérance aux pannes :

- le concept de **point de reprise** (ou Checkpoint) est une abstraction des points de reprise qui est utilisée par FT-GReLoSSS. Cette abstraction réalise les E/S des points de reprise séquentiels et fournit une interface de contrôle des points de reprise. Notamment elle permet au programmeur d'établir la fréquence de réalisation des points de reprise mais également de sélectionner les données à inclure dans le point de reprise. La fréquence de réalisation des points de reprise est fonction du nombre d'itérations dans la boucle de calcul ;
- le **gestionnaire de tolérance aux pannes** maintient des informations relatives à la tolérance aux pannes. Des informations de configuration par exemple sur l'emplacement de stockage des points de reprise. Il maintient l'état de l'application en indiquant si l'on se trouve en exécution normale ou en phase de reprise sur panne. Pendant la reprise, il est responsable de la détermination de la ligne de reprise la plus récente. C'est un composant présent sur chaque processus de l'application et auquel on peut potentiellement assigner d'autres tâches liées à la tolérance aux pannes.

Les squelettes tolérants aux pannes sont obtenus en définissant des emplacements de points de reprise aux sein de la boucle principale à l'aide du concept de point de reprise dont FT-GReLoSSS fournit une abstraction. La figure 5.10 représente la version tolérante aux pannes du squelette SPMD à nombre d'étapes connu de SPMD qui est présent à la figure 5.2 (p. 70). Le squelette tolérant aux pannes comporte la déclaration d'un point de reprise (l. 4) qui est placé après l'opération de `swap` (l. 14). La méthode `run` rend effectif le point de reprise. Par

```
1  template<typename T_numtype, int N_rank>
2  class     Matmult_Domain: public Domain<double, 2, Matmult_Domain>
3  {
4  private:
5       blitz::Array<double, 2>    data;
6
7  public:
8       Matmult_Domain(int rank, int numprocs, TinyVector<int, 2> extent):
9            // Appel au constructeur de la classe mère pour bien initialiser.
10           Domain<double, 2, ::Matmult_Domain>(rank, numprocs, extent)
11      {
12           Domain_desc<2> dd = data_needed(rank, numprocs, 0);
13           data.resize(dd.extent(1), dd.extent(2));
14      }
15
16      Domain_desc<2> data_needed(int rank, int numprocs, int step)
17      {
18           int size = this->get_extent(blitz::firstDim);
19           int partition_size = size / numprocs;
20
21           int dim1_lbound, dim1_rbound;
22
23           // Calcul des frontières du domaine
24           dim1_lbound = (rank + step)%numprocs * partition_size;
25           dim1_rbound = dim1_lbound + partition_size - 1;
26
27
28           Domain_desc<2>  domain_desc;
29           domain_desc.set_bounds(1, dim1_lbound, dim1_rbound);
30           domain_desc.set_bounds(2, 0, size-1);
31
32           return domain_desc;
33      }
34
35      Domain_desc<2> data_possessed(int rank, int numprocs, int step)
36      {
37           return data_needed(rank, numprocs, step);
38      }
39
40      double lget(blitz::TinyVector<int, 2>  &coord)
41      {
42           return data(coord(0), coord(1));
43      }
44
45      void lset(blitz::TinyVector<int, 2> &coord, double e)
46      {
47           data(coord(0), coord(1)) = e;
48      }
49
50      void swap(Matmult_Domain<double, 2>    *md)
51      {
52           blitz::cycleArrays(this->data, md->get_data());
53      }
54 }
```

FIGURE 5.7 – Définition de la structure de données distribuée de *Matmult*.

```
1  class   Matmult_Kernel: public FT_Calc_Kernel
2  {
3       // Définition du domaine.
4       Matmult_Domain<double, 2>   A1, // Calc. Read Buffer
5                                   A2; // Comm. Write Buffer
6
7       Array<double, 2>  TB,  // Bloc local fixe de la transposée de la matrice B.
8                         C;   // Bloc local fixe de la matrice résultat C.
9
10      // Constructeur.
11      Matmult_Kernel(int myid, int numprocs, TinyVector<int, 2> extent):
12           myid(myid),
13           numprocs(numprocs),
14           A1(myid, numprocs, extent),
15           A2(myid, numprocs, extent),
16           size(extent(0)),
17           local_size(extent(0)/numprocs),
18           TB(local_size, size),
19           C(size, local_size)
20      {
21           // Méthode privée qui initialise A1, A2, B et C
22           LocalMatrixInit();
23      }
24
25      //   Méthode de calcul.
26      void compute()
27      {
28           int   i, j, k;
29           int   OffsetLigneC;
30
31           // At step "step", the processor compute the C block starting at line:
32           // ((myid+step)*local_size)%size
33           OffsetLigneC = ((myid + A1.get_step()) * local_size) % size;
34           for (i = 0; i < local_size; ++i)
35                for (j = 0; j < local_size; ++j)
36                     for (k = 0; k < size; ++k)
37                          C(i + OffsetLigneC, j) += A1.get(i, k) * TB(j, k);
38      }
39 };
```

FIGURE 5.8 – Définition du noyau de calcul de *Matmult*.

```
1  int main(int argc, char **argv)
2  {
3       // Initialisations ---------------------------------//
4
5       //   + Initialisations liées à MPI.
6       MPI_Init(&argc, &argv)
7       // ...
8
9       //   + Init. du gestionnaire de tolérance aux pannes de FT-GReLoSSS.
10      FT_Mgr::init(&argc, &argv);
11
12      //   + Init. du noyau de calcul.
13      TinyVector<int, 2> extent(size, size);     // Etendue de chaque dimension
14                                                 // des matrices
15      Matmult_Kernel<double, 2, Matmult_Domain> mk(extent);
16
17      //   + Init. du squelette avec le noyau de calcul
18      FT_skel<double, 2, Matmult_Domain>
19           Matmult_FT_Skel(&mk,
20                           &mk.A1,    // Calc. Read buffer
21                           &mk.A2);   // Comm. Write buffer
22
23      // Lancement du squelette--------------------------//
24      Matmult_FT_Skel.execute();
25
26      // Nettoyage de FT-GReLoSSS-----------------------//
27      FT_Mgr::finalize();
28
29      MPI_Finalize();
30
31 } // FIN du  main()
```

FIGURE 5.9 – Fonction principale de l'application *Matmult*.

ailleurs, au lieu des fonctions `compute` et `comms`, le squelette utilise leur équivalents tolérants aux pannes `ft_compute` et `ft_comms` (resp. l. 11 et l.12). Les fonctions préfixées de `ft_` sont des fonctions emballages qui tiennent compte du mode d'exécution (normal ou reprise) tel qu'il est décrit par MoLOToF et permettent d'assurer une reprise sur panne rapide (cf. Section 4.2.2). Le gestionnaire de tolérance aux pannes intervient dans ces fonctions `ft_` et dans l'abstraction de point de reprise de FT-GReLoSSS en fournissant l'information sur l'état de l'application (mode normal ou reprise). Le squelette à nombre d'itérations inconnu est modifié similairement (cf. Figure 5.11).

Compte-tenu de la nature SPMD des programmes FT-GReLoSSS, tous les processus réalisent leurs points de reprise au même emplacement dans le code et au même moment conformément à la période de réalisation de points de reprise spécifiée par le programmeur. Cependant, ceci ne suffit pas pour que les points de reprise réalisés forment une ligne de reprise. L'approche adoptée dans FT-GReLoSSS est de garantir l'absence de communications au moment de la réalisation des points de reprise. FT-GReLoSSS y parvient grâce au plan de routage qui contrôle les communications et assure qu'à la fin de son exécution toutes ont été achevées. En ce qui concerne les communications éventuelles dans la fonction de test de la condition d'arrêt (dans le cadre d'un squelette à nombre d'itérations inconnu), elles sont entièrement réalisées dans la fonction de test. Ainsi, l'absence de communications au moment de la réalisation du point de reprise est respectée. Le programmeur sera néanmoins peut être amené à enregistrer auprès du point de reprise, les valeurs de certaines variables utilisées dans la fonction de test.

```
1  class FT_GReLoSSS_Skel_for
2  {
3       // ...
4       Checkpoint c;
5
6       void execute()
7       {
8            // ...
9            for (it = sfi.beg(); it != sfi.end(); it = sfi.next())
10           {
11                ft_compute(sfi);     // Phase de calculs
12                rp->ft_comms(sfi);   // Phase de communications
13                swap();              // Echange des tableaux à N dimensions
14                c.run(it);           // Point de reprise possible
15           }
16      }
17 };
```

FIGURE 5.10 – Squelette tolérant aux pannes de FT-GReLoSSS à nombre d'étapes connu.

5.4.1 Implication du programmeur dans FT-GReLoSSS

Le programmeur revêt un rôle important dans le développement d'applications tolérantes aux pannes avec FT-GReLoSSS. Il exerce ce rôle à travers l'interface fournie par FT-GReLoSSS qui (lui) permet de contrôler différents aspects liés à la tolérance aux pannes.

En accord avec les principes du modèle MoLOToF (cf. Section 4.2.3), les squelettes tolérants aux pannes de FT-GReLoSSS sauvent l'itérateur de la boucle tolérante aux pannes, l'identifiant

```
1  class FT_GReLoSSS_Skel_while
2  {
3       // ...
4       Checkpoint c;
5
6       void execute()
7       {
8            // ...
9            while ((it1 != sfi.end()) && (it2 != swi.end()))
10           {
11                ft_compute(sfi);      // Phase de calculs
12                rp->ft_comms(sfi);    // Phase de communications
13                swap();               // Echange des tableaux à N dimension
14                c.run(it);            // Point de reprise possible
15                it1 = sfi.next();
16                it2 = swi.next();
17           }
18      }
19 };
```

FIGURE 5.11 – Squelette tolérant aux pannes de FT-GReLoSSS à nombre d'étapes inconnu.

du squelette mais également les deux tableaux à N dimensions du modèle. Le programmeur a la possibilité de débrayer la sauvegarde de l'une ou des deux tableaux à N dimensions mais en veillant à conserver un point de reprise correct. Les autres données nécessitant d'être sauvées doivent être renseignées par le programmeur dans le cadre de la *collaboration de correction* (cf. Section 4.2.3). Ceci correspond aux variables modifiées par la fonction de calcul mais non locales à cette dernière. La section 5.4.2 fournit un exemple de collaboration de correction dans le cadre de l'implantation, avec FT-GReLoSSS, d'une application réalisant un produit de matrices denses en parallèle. Enfin, FT-GReLoSSS permet au programmeur de régler la fréquence de réalisation des points de reprise dans le cadre de la *collaboration de fréquence* (cf. Section 4.2.3). Un exemple de ce type de collaboration est également présent à la section 5.4.2.

5.4.2 Exemple de tolérance aux pannes avec *Matmult*

L'application *Matmult* initialement décrite à la section 5.3.4 réalise un produit de matrices denses en parallèle sur un anneau de processus. Le résultat de l'analyse en termes de données lues et écrites qui est présent à la figure 5.6 (p. 74) facilite l'application de la tolérance aux pannes.

À chaque étape du calcul, un nouveau bloc résultat est écrit dans le tableau résultat. Pour assurer la correction du point de reprise, il est donc impératif de sauver le tableau résultat : c'est ce qui est fait dans l'extrait de code de la fonction principale de l'application (Figure 5.12, l. 34-35)) et où la variable C correspond au tableau résultat. En revanche, le bloc de matrice B est toujours en lecture et n'a pas besoin d'être enregistré auprès du point de reprise : il sera restitué à la reprise par ré-exécution d'un code d'initialisation. Quant à la période de réalisation de points de reprise, elle est précisée en paramètre du constructeur du squelette (Figure 5.12, l. 21).

Enfin, au sein du squelette, étant donné l'emplacement de l'instruction de réalisation de point de reprise, on se rend compte qu'il est inutile de conserver *shadow* (cf. Figure 5.6) dans le point

de reprise. En effet, comme le point de reprise est réalisé après l'opération d'échange (`SWAP`) les données qui seront utilisées se trouvent dans la tableau en lecture. Ainsi, *shadow* est retiré du point de reprise. Cette opération est effectuée à la ligne 28 de la figure 5.12. La variable `write_buffer` sur laquelle porte l'opération est la même que *shadow* à la figure 5.6.

```
1  int main(int argc, char **argv)
2  {
3       // Initialisations ------------------------------//
4
5       //   + Initialisations liées à MPI.
6       MPI_Init(&argc, &argv);
7       // ...
8
9       //   + Init. du gestionnaire de tolérance aux pannes de FT-GReLoSSS.
10      FT_Mgr::init(&argc, &argv);
11
12      //   + Init. du noyau de calcul.
13      TinyVector<int, 2> extent(size, size);     // Etendue de chaque dimension
14                                                 // des matrices
15      Matmult_Kernel<double, 2, Matmult_Domain> mk(extent);
16
17      //   + Init. du squelette avec le noyau de calcul
18      FT_skel<double, 2, Matmult_Domain>
19           Matmult_FT_Skel(&mk,
20                           &mk.A1,  // Calc. Read buffer
21                           &mk.A2,  // Comm. Write buffer
22                           checkpoint_period); // Période de réalisation des
23                                               // points de reprise
24
25      // Quelques réglages de tolérance aux pannes ------------------- //
26
27      //   + Optimisation de la taille des points de reprise: désenregistrement du tampon
28      //     en écriture.
29      Matmult_FT_Skel.do_unregister_var(WRITE_BUFFER);
30
31      //   + Addition de la matrice résultat C au point de reprise
32      //        + C->dataFirst(): adresse du premier élément du tableau à N dimensions
33      //        + C->numElems(): nombre d'éléments dans le tableau
34      //        + PRECONDITION: les éléments doivent être contigus en mémoire.
35      Array<double, 2> *C = mk.get_C();
36      Matmult_FT_Skel.do_register_var(C->dataFirst(), C->numElems());
37
38      // Lancement du squelette------------------------------//
39      Matmult_FT_Skel.execute();
40
41      // Nettoyage de FT-GReLoSSS----------------------------//
42      FT_Mgr::finalize();
43
44      MPI_Finalize();
45
46 } // FIN du  main()
```

FIGURE 5.12 – Fonction principale de l'application *Matmult* avec tolérance aux pannes.

5.4.3 Exemple de tolérance aux pannes avec *Jacobi*

L'application *Jacobi* réalise une relaxation de Jacobi sur un domaine bi-dimensionnel qui est partagé selon une seule dimension entre P processus. La figure 5.13 illustre un tel découpage dans le cas d'un partage entre 3 processus d'une grille de taille utile $N = 8$. Dans la partie (ou sous-domaine) qui lui est affectée, un processus met à jour la valeur de chaque point comme

étant la moyenne arithmétique des valeurs de ses quatre voisins. Les valeurs initiales et les valeurs mises à jour sont stockées dans des tableaux différents. Le calcul se poursuit jusqu'à atteindre la convergence de la solution représentée par l'ensemble des valeurs stockées.

La mise en œuvre de cet algorithme avec FT-GReLoSSS est directe et s'effectue en utilisant deux tableaux à 2 dimensions (appelés également grilles) (cf. figure 5.14). Pendant la phase de calcul, la première grille qui contient les valeurs au pas de temps précédent $i-1$, est lue pour calculer les valeurs au pas de temps courant i. Les nouvelles valeurs sont stockées dans la seconde grille. La première et la seconde grille correspondent respectivement aux tableaux *NDArray 1* et *NDArray 2* de la figure 5.4 qui illustre le modèle de parallélisation de FT-GReLoSSS.

Une analyse des variables en lecture et en écriture conduit à ne conserver dans le point de reprise que la seconde grille. En effet, après l'opération d'échange (*i.e.* : `SWAP`), seule la première grille contient les données nécessaires à la prochaine itération. Contrairement à l'application *Matmult*, l'application *Jacobi* ne nécessite pas la sauvegarde de données supplémentaires en dehors des données déjà sauvées par le squelette. Un extrait de la fonction principale de l'application *Jacobi* est présent à la figure 5.15. Ce code est très similaire à celui de la fonction principale de l'application *Matmult* (cf. Figure 5.12). Nous y retrouvons l'établissement de la période de réalisation de points de reprise (l. 22) et la suppression de la deuxième grille du point de reprise (l. 29).

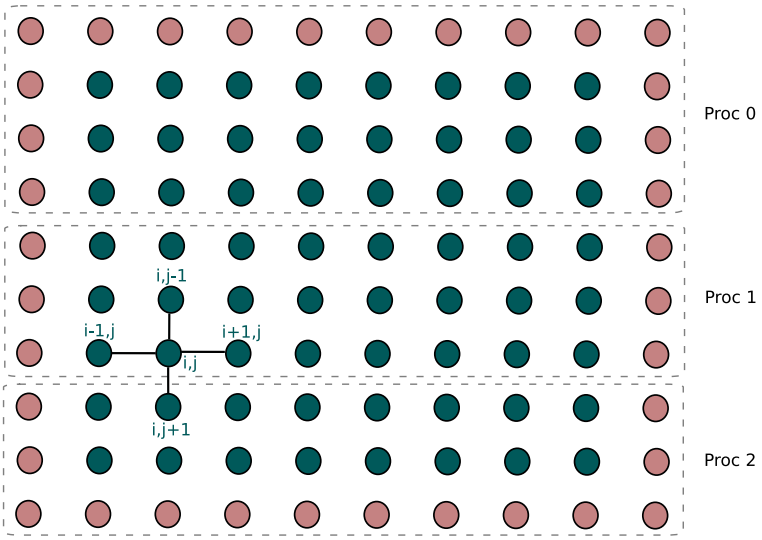


FIGURE 5.13 – Décomposition 1D d'une grille carrée de taille utile $N = 8$ parmi 3 processus.

5.5 Protocole piloté de réalisation de points de reprise de FT-GReLoSSS

FT-GReLoSSS est prévu pour avoir également un fonctionnement piloté tel qu'il est préconisé dans le modèle MoLOToF (cf. Section 4.2.4) dans le cadre d'une collaboration du SPRD avec l'environnement d'exécution.

Pour les applications selon le modèle SPMD, l'approche utilisée consiste à rajouter une opération qui s'enquiert auprès d'un service centralisé de la nécessité de réaliser une opération de sauvegarde. Cette opération se trouve au niveau des emplacements de réalisation de points de reprise. Elle fait parti de la condition de réalisation de points de reprise et elle est exécutée à

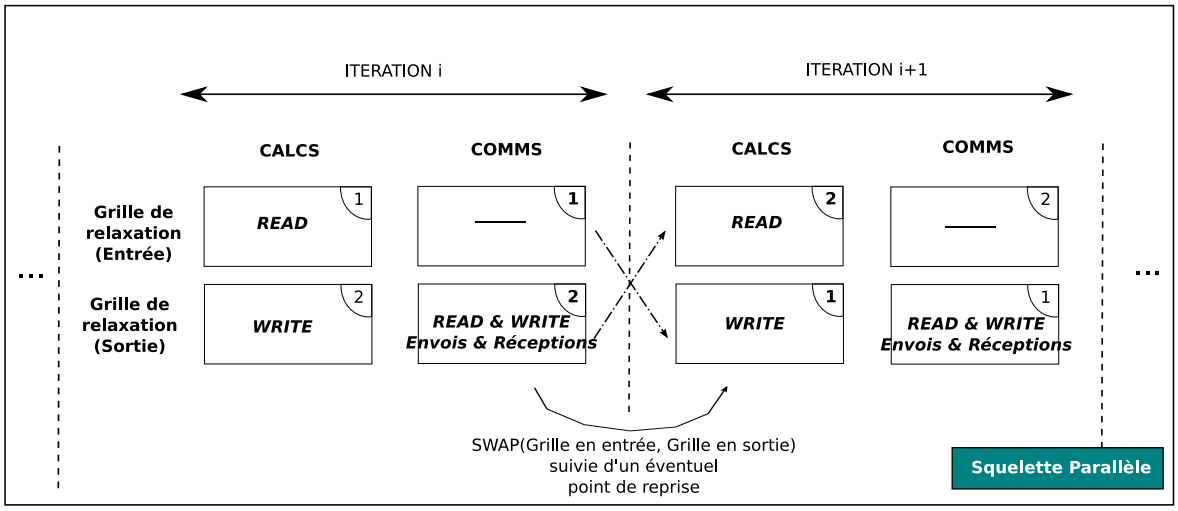


FIGURE 5.14 – Adaptation d'un algorithme de relaxation de Jacobi parallèle au modèle de parallélisation de FT-GReLoSSS.

chaque passage par l'emplacement de réalisation de points de reprise. Cette opération comprend, pour chaque processus applicatif, l'envoi d'un message pour demander s'il faut réaliser une sauvegarde et la réception d'un message correspondant à la réponse de la demande. Il en découle que, pour une application de N processus, $2 \times N$ messages sont échangés à chaque passage par un tel emplacement. Cette approche est très facile à mettre en place. Nous la retrouvons par exemple au centre de calcul intensif de Pittsburgh [144] (cf. appel `tcs_drain_operation_`). C'est l'approche qui est également adoptée par le framework SRS [153] (cf. appel `SRS_Check_Stop`) initialement développé à l'université du Tennessee.

Nous proposons ci-après un protocole qui vise à limiter ces communications réseau systématiques. Le nombre de communications lors des interactions avec l'environnement est réduit lorsque le nombre de requêtes est *faible* par rapport au nombre de super-étapes. La mise en œuvre de ce protocole s'appuie sur et est rendue possible par les squelettes tolérants aux pannes (de FT-GReLoSSS).

5.5.1 Principes du protocole

Dans le protocole utilisé par défaut par les squelettes tolérants aux pannes, les processus connaissent préalablement les endroits de réalisation des points de reprise. Suite à une requête externe, les processus doivent s'arranger pour réaliser leurs sauvegardes individuelles aux mêmes emplacements de réalisation de points reprise ; et ce, afin que le point de reprise distribué résultant soit cohérent. Pour y parvenir, le protocole que nous proposons fait intervenir, en plus des **processus applicatifs**, des **processus de contrôle**. Le terme de processus est employé ici avec un sens large et désigne une entité réalisant un traitement et pouvant à l'implantation se matérialiser par des processus lourds ou légers (threads). Un processus applicatif est un processus responsable de l'exécution des parties utilisateur de l'application de calcul distribué (*i.e.* : calculs et communications). À chaque processus applicatif est associé un processus de contrôle. Ce dernier est chargé des aspects de coordination pour la cohérence des points de reprise réalisés. L'idée sous-jacente est que les processus applicatifs fournissent aux processus de contrôle une information caractérisant le **point d'avancement** dans le calcul. Les processus de contrôle se servent alors de cette information pour se coordonner. Ceci conduit d'une part à une synchronisation entre

```
1  int main(int argc, char **argv)
2  {
3       // Initialisations ----------------------------------//
4
5       //   + Initialisations liées à MPI.
6       MPI_Init(&argc, &argv);
7       // ...
8
9       //   + Init. du gestionnaire de tolérance aux pannes de FT-GReLoSSS.
10      FT_Mgr::init(&argc, &argv);
11
12      //   + Init. du noyau de calcul.
13      TinyVector<int, 2> extent(size, size);     // Etendue de chaque dimension
14                                                 // des grilles de relaxation
15      Jacobi_Kernel<double, 2, Jacobi_Domain> jk(extent, vmax, nbcycles);
16
17      //   + Init. du squelette avec le noyau de calcul
18      FT_skel<double, 2, Jacobi_Domain>
19           Jacobi_FT_Skel(&jk,
20                          &jk.calc_read_buffer,
21                          &jk.comm_write_buffer,
22                          checkpoint_period); // Période de réalisation des
23                                              // points de reprise
24
25      // Quelques réglages de tolérance aux pannes ------------------ //
26
27      //   + Optimisation de la taille des points de reprise: désenregistrement du tampon
28      //     en écriture.
29      Jacobi_FT_Skel.do_unregister_var(WRITE_BUFFER);
30
31      // Lancement du squelette-------------------------------------//
32      Jacobi_FT_Skel.execute();
33
34      // Nettoyage de FT-GReLoSSS----------------------------------//
35      FT_Mgr::finalize();
36
37      MPI_Finalize();
38
39 } // FIN du  main()
```

FIGURE 5.15 – Fonction principale de l'application *Jacobi* avec tolérance aux pannes.

les processus de contrôle ; et d'autre part à une synchronisation entre un processus de contrôle et son processus applicatif associé.

5.5.1.1 Synchronisation entre processus de contrôle

À la manière des bibliothèques de communication MPI, chaque processus de contrôle possède un rang qui lui est attribué au déploiement de l'application et qui permet de l'identifier de manière unique au sein de l'application. Parmi les processus de contrôle, il en existe un qui va jouer le rôle de coordinateur : dans notre cas il s'agit du processus de rang 0.

À la réception d'une requête de réalisation de point de reprise, le coordinateur diffuse la requête à tous les autres processus intervenant dans le calcul. Cette requête contient une paire d'identifiants : l'identifiant du squelette et l'identifiant de l'itération en cours $((skel\_id, iter\_id))$. Cette paire d'identifiants est suffisante pour caractériser *le point d'exécution* du processus applicatif : nous l'appelons *identifiant d'avancement*. Le coordinateur se met ensuite en attente des identifiants d'avancement des autres processus. À la réception de l'identifiant d'avancement du coordinateur, les autres processus de contrôle se contentent de lui envoyer le leur. Il se mettent ensuite en attente de la réponse du coordinateur. Lorsque le coordinateur a récupéré tous les identifiants, il diffuse aux processus de contrôle l'identifiant le plus avancé. Ces derniers utilisent cette information afin que leurs processus applicatifs respectifs réalisent une sauvegarde à l'emplacement de réalisation de point de reprise suivant (lors de la prochaine mise à jour du point d'avancement).

5.5.1.2 Synchronisation entre processus de contrôle et processus applicatif

Pendant l'exécution, chaque processus applicatif tient informé son processus de contrôle de l'identifiant d'avancement : à chaque passage par un emplacement de point de reprise, le processus applicatif notifie le processus de contrôle du nouvel identifiant d'avancement. Il demande également s'il doit effectuer un point de reprise dans le cadre d'une requête externe. Ces deux opérations s'effectuent lors du passage par le point de reprise. En absence de traitement de requête externe, le processus de contrôle accepte la mise à jour de l'identifiant d'avancement et répond négativement à la demande de réalisation de point de reprise. En présence de traitement d'une requête externe, le processus de contrôle accepte la mise à jour et fait patienter le processus applicatif jusqu'à ce qu'il sache à quel point d'avancement le point de reprise doit être réalisé.

5.5.2 Exemple de cas favorable

La figure 5.16 illustre une exécution du protocole en présence de trois processus applicatifs qui traversent les emplacements de points de reprise `C11`, `C12` et `C13`. Dans `Cij`, `i` correspond à l'identifiant du squelette et `j` à celui de l'itération en cours. Les processus applicatifs sont dénotés `Appl_PRk` et les processus de contrôle associés sont dénotés `Ctrl_PRk`. `k` varie de 1 à 3 et permet d'identifier les processus. En particulier, le processus de contrôle `Ctrl_PR1` revêt le rôle de coordinateur. Ce dernier reçoit de l'environnement une requête de réalisation de point de reprise `CR` après l'emplacement du point de reprise `C12`. L'étape de coordination avec les autres processus de contrôle est initiée en diffusant `C12` à `Ctrl_PR2` et `Ctrl_PR3` qui répondent tous les deux par `C12` et se mettent en attente de la réponse du coordinateur. Ce dernier met fin à la coordination en diffusant le point d'avancement le plus avancé (`C12`) : tous les processus applicatifs réaliseront donc un point de reprise en `C13`. Cette exécution est très favorable car la synchronisation des processus de contrôle se déroule entièrement entre deux emplacements de

points de reprise : on réalise un recouvrement total entre la synchronisation et l'exécution des processus applicatifs.

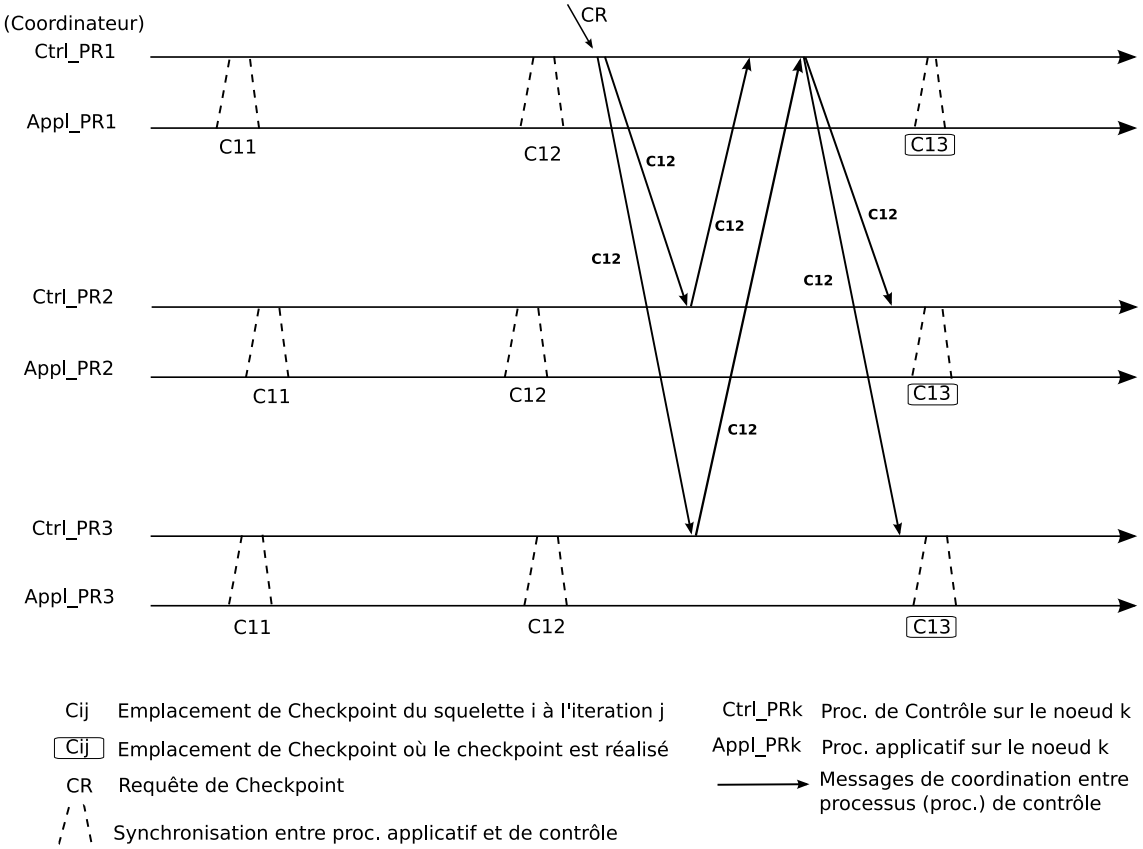
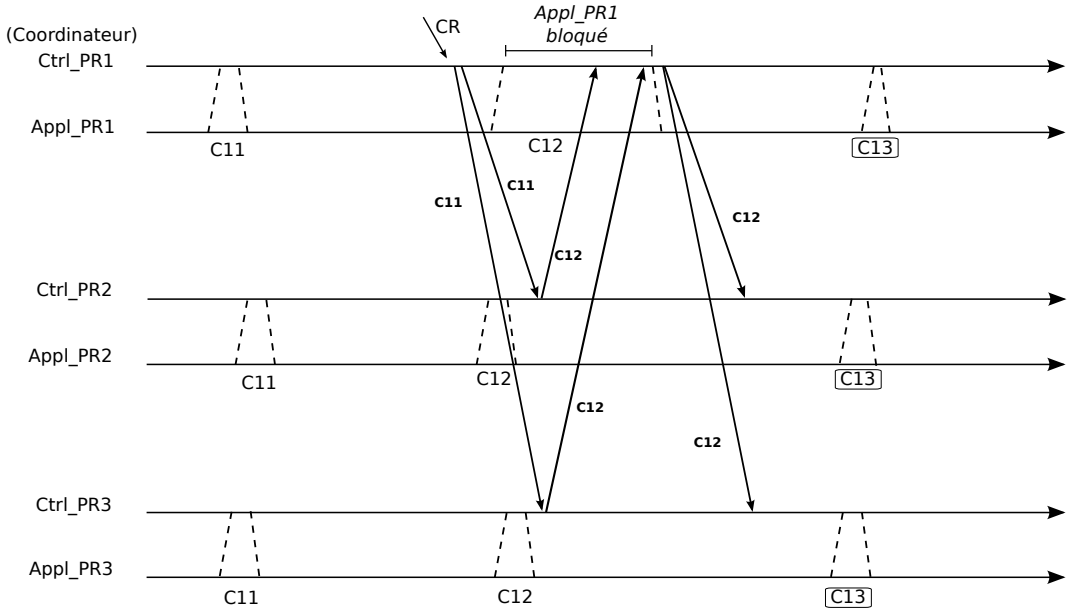


FIGURE 5.16 – Protocole piloté de FT-GReLoSSS : cas favorable.

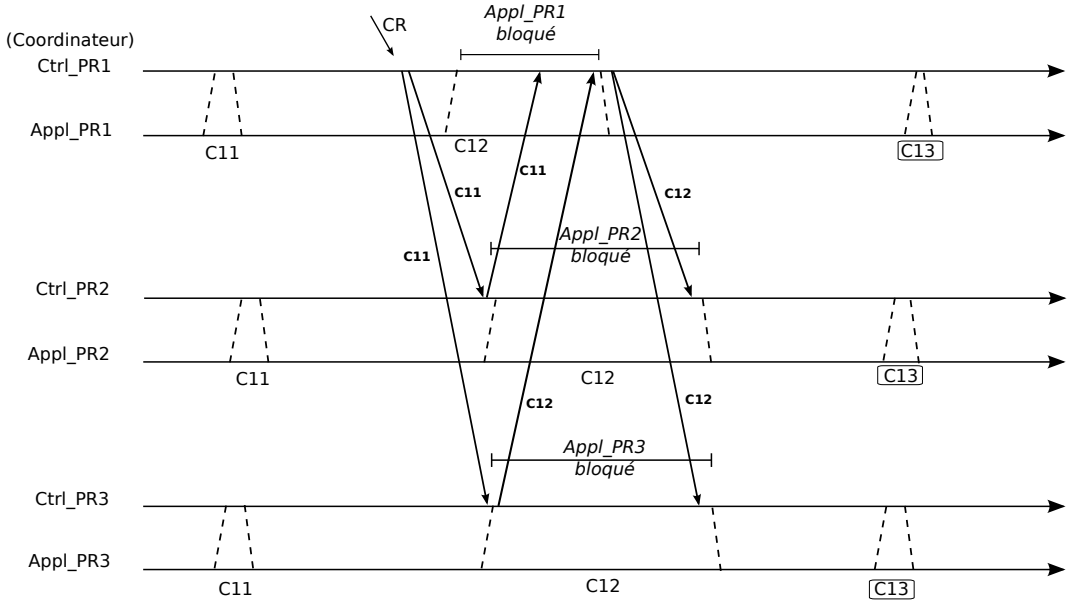
5.5.3 Exemples de cas défavorables

La figure 5.17a illustre un cas un peu moins favorable dans la mesure où les processus de contrôle `Ctrl_PR2` et `Ctrl_PR3` sont informés de la requête de réalisation de point de reprise après que leur processus applicatifs respectifs `Appl_PR2` et `Appl_PR3` ont franchi l'emplacement de réalisation de point de reprise `C12` (ou point de synchronisation). Le processus applicatif `Appl_PR1` se trouve alors bloqué à attendre une réponse de son processus de contrôle `Ctrl_PR1`. Ce dernier ne peut débloquer `Appl_PR1` que lorsqu'il reçoit les réponses des processus `Ctrl_PR2` et `Ctrl_PR3`. Après réception des réponses, `Ctrl_PR1` connaît l'emplacement définitif de réalisation du point de reprise et le diffuse aux autres processus de contrôle.

La figure 5.17b illustre un cas défavorable un peu plus complexe que celui présent à la figure 5.17a. Dans le cas considéré à la figure 5.17b la requête de réalisation de point de reprise est receptionnée par tous les processus applicatifs un peu avant le franchissement de l'emplacement `C12`. Les processus applicatifs `Appl_PR1`, `Appl_PR2` et `Appl_PR3` doivent ainsi rester bloqués dans `C12` juqu'à la fin de la synchronisation des processus de contrôle avant de pouvoir reprendre leur exécution. Le temps de bloquage correspond approximativement au temps nécessaire à la coordination entre les processus de contrôle. Le cas est défavorable car il n'y a pas de recouvrement de la synchronisation avec les calculs. À la fin de la synchronisation, les processus applicatifs



(a) Cas défavorable 1



(b) Cas défavorable 2

FIGURE 5.17 – Illustration de quelques cas défavorables pour le protocole piloté de FT-GReLoSSS.

reprennent leur exécution : il réaliseront un point de reprise en `C13`.

Dans les deux scénarios nous pouvons voir une source potentielle de ralentissement liée au bloquage des processus applicatifs au niveau des emplacements de points de reprise. Nous proposons dans la section suivante quelques optimisations qui utilisent la valeur du point d'avancement pour limiter les bloquages.

5.5.4 Optimisations du protocole

Le protocole sous sa forme présentée à la section 5.5.1 peut conduire à des bloquages dont nous avons rencontré certains à la section 5.5.3. Un bloquage intervient dans la synchronisation entre un processus applicatif et son processus de contrôle lorsque le processus applicatif effectue sa mise à jour du point d'avancement et que le processus de contrôle est en présence de traitement d'une requête externe.

L'idée derrière les optimisations proposées est d'exploiter la valeur des points d'avancement de manière à réduire le nombre de bloquages. Par exemple, dans le cas défavorable de la figure 5.17a, le coordinateur reste bloqué jusqu'à la réception de la réponse de `Ctrl_PR3`. Or, peu de temps avant il vient de recevoir la réponse de `Ctrl_PR2` qui correspond à un `C12`. En d'autres termes, `Appl_PR2` (le processus applicatif de `Ctrl_PR2`) n'est pas bloqué au point `C12`. En conséquence, le point de reprise ne pourra pas avoir lieu en `C12`. Il est donc inutile de maintenir `Appl_PR1` davantage bloqué. Il peut être débloqué dès la réception de la réponse de `Ctrl_PR2` (cf. Figure 5.18).

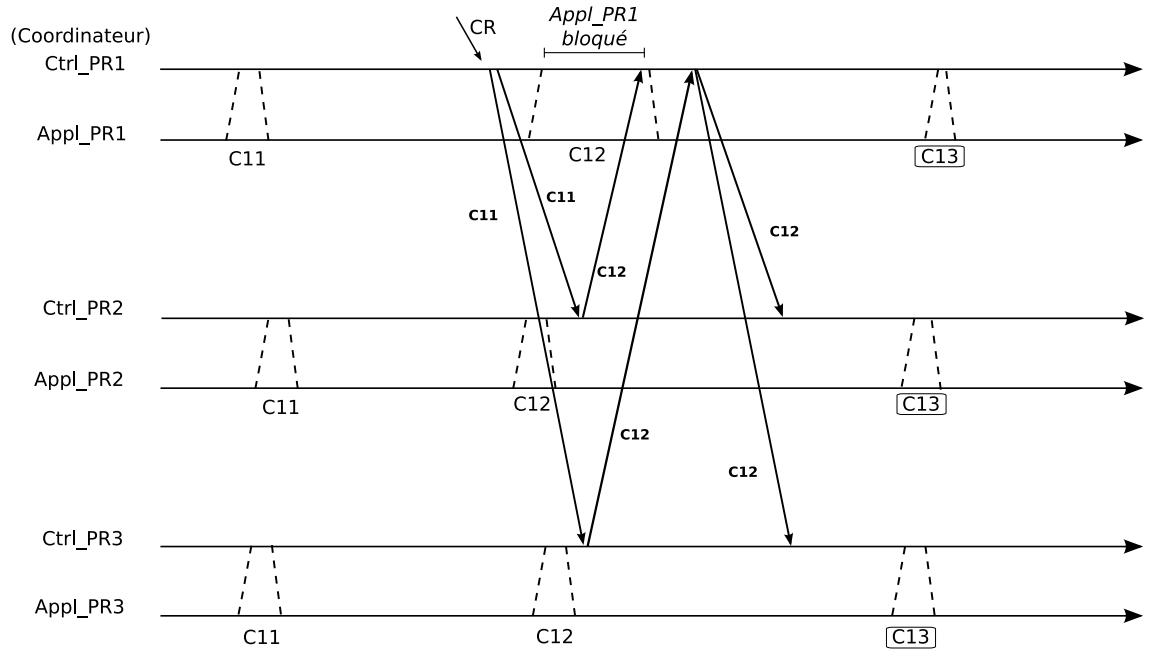


FIGURE 5.18 – Protocole piloté de FT-GReLoSSS : optimisation en action sur le coordinateur dans le cadre du cas défavorable 1 (cf. Figure 5.17a).

La même optimisation peut être utilisée au niveau des processus de contrôle autres que le coordinateur. En effet, supposons qu'un processus de contrôle `Ctrl_PRk` avec un point d'avancement courant `C10` reçoive de la part du coordinateur un point d'avancement `C11`. `Ctrl_PRk` sait alors qu'il est inutile de bloquer son processus applicatif `Appl_PRk` au point `C11` car le processus applicatif du coordinateur est au-delà de `C11`. Donc, le point de reprise ne pourra être réalisé

avant `C12`. `Appl_PRk`. Ainsi, si `Ctrl_PRk` n'a toujours pas reçu la réponse du coordinateur d'ici l'arrivée de `Appl_PRk` à `C12`, `Appl_PRk` sera bloqué.

À ce stade, nous pouvons remarquer l'asymétrie entre le coordinateur et les autres processus de contrôle. En effet, le coordinateur reçoit les points d'avancement de tous les processus de contrôle tandis que les processus de contrôle (autres que le coordinateur) ne reçoivent que le point d'avancement du coordinateur. Il en découle que l'optimisation n'aura pas le même effet sur chaque processus de contrôle : elle dépendra du point d'avancement courant. Pour ce qui est du coordinateur, le fait de prendre connaissance des points d'avancements de tous les processus de contrôle limite davantage la possibilité de bloquage mais reste dépendant de l'ordre d'arrivée des réponses des processus de contrôle.

5.5.5 Limites du protocole piloté

Le protocole décrit dans les sections précédentes fait intervenir deux types de processus : applicatifs et de contrôle. Chaque processus applicatif est apparié avec un processus de contrôle. Les processus de chaque paire se surveillent mutuellement de sorte qu'une action appropriée puisse être entreprise en cas de détection de pannes. Actuellement, la stratégie utilisée est de mettre fin à l'exécution de l'application à la moindre détection de panne du processus applicatif ou de contrôle, et de relancer l'application le plus vite possible sur un système distribué sain.

La synchronisation entre les processus de contrôle met en jeu $3 \times (N-1)$ messages à chaque requête de l'environnement : $N-1$ messages pendant la phase de diffusion de la requête, $N-1$ messages pour l'envoi de la réponse des $N-1$ processus de contrôle au coordinateur et enfin $N-1$ messages pour la confirmation. En comparaison, le protocole employé dans [153, 144] que nous appelerons *protocole standard* est indépendant du nombre de requêtes : il échange $2 \times N$ messages à chaque passage par un emplacement de réalisation de points de reprise. Ainsi, si l'application passe par M emplacements de réalisation de points de reprise et reçoit R requêtes de réalisation de points de reprise on aura :

- $2 \times NM$ messages échangés dans le protocole standard et
- $3 \times (N-1)R$ messages échangés dans le protocole piloté de FT-GReLoSSS.

Le nombre de messages échangés dans le protocole de FT-GReLoSSS est inférieur à celui dans le protocole standard lorsque $2NM > 3(N-1)R$ ce qui revient à $\frac{2NM}{3(N-1)} > R$. Pour N grand, l'inégalité se simplifie en $\frac{2}{3}M > R$. En d'autres termes, si le nombre de requêtes est inférieur au $\frac{2}{3}$ du nombre d'emplacements de points de reprise, notre protocole échange moins de messages que le protocole standard. Au delà, il en échange plus mais ceci est caché en parti ou en totalité grâce au recouvrement réalisé. En effet, une originalité de notre protocole est de tirer parti des créneaux entre les emplacements de points de reprise pour recouvrir les communications de synchronisation avec les phases de calcul ou communication de l'application.

Pour le moment, ce protocole n'a pas fait l'objet d'une quelconque validation pratique ou formelle au cours desquelles de nouvelles limitations pourraient apparaître. Une implantation est en cours de réalisation. Elle constituera une première étape dans la validation et permettera d'évaluer la performance pratique du protocole. Enfin, le protocole piloté présenté dans cette section est compatible avec le protocole par défaut de FT-GReLoSSS. L'implantation du protocole piloté permettra d'étudier comment se déroule la cohéxistence et peut être dégager des réglages de fonctionnement intéressants.

5.6 Détails d'implantation de FT-GReLoSSS

FT-GReLoSSS est implanté en langage C++ et il utilise la bibliothèque de communication *Open MPI* pour réaliser les communications applicatives. De part son architecture, qui repose sur la spécification MPI, il peut être associé à n'importe quelle autre implantation MPI sans changement. Ceci est intéressant pour l'exécution sur des environnements spécialisés comme ceux de certains supercalculateurs qui possédent leur propre implantation de MPI afin de bénéficier d'optimisations spécifiques au matériel utilisé. Pour l'écriture des points de reprise sur disque, FT-GReLoSSS permet l'écriture en mode binaire standard ou avec la bibliothèque *NetCDF* [99]. Cette dernière, très répandue en calcul scientifique, permet de réaliser des points de reprise portables entre architectures différentes. Enfin, FT-GReLoSSS utilise également les bibliothèques *Blitz++* [13] et *Boost* [14] qui sont des standards très utilisés.

5.6.1 Tableaux distribués à N dimensions

Le concept de tableau distribué n'est pas neuf dans le monde du parallélisme et a été utilisé dans plusieurs travaux de recherche [97, 78]. FT-GReLoSSS utilise également comme structure de données distribuée des tableaux à N dimensions. Mais, à notre connaissance, FT-GReLoSSS est le seul à ne pas imposer la structure sous-jacente : en effet, FT-GReLoSSS offre une interface pour tableaux à N dimensions. FT-GReLoSSS suppose que le tableau possède une représentation physique à une dimension. C'est le cas de la plupart des implantations existantes.

L'utilisation de l'héritage classique pour mettre en œuvre une telle interface est source d'un surcoût en performance important étant donné que l'accès à un élément n'est pas direct et connaît une indirection. Pour pallier cet inconvénient, FT-GReLoSSS utilise le *Curiously Recurring Template Pattern (CRTP)* attribué à [37]. Une application de ce pattern est de réaliser du polymorphisme similaire à celui de l'héritage classique mais de manière statique. Il en résulte l'élimination des problèmes de performance occasionnés par l'accès aux éléments.

Chaque processus peut accéder de deux manières aux éléments de la partie du tableau à N dimensions qui lui a été confiée (ou tableau local). La première consiste à utiliser les *coordonnées globales* c'est-à-dire les mêmes coordonnées qu'il aurait utilisées s'il avait en sa possession tout le tableau. La seconde consiste à utiliser les *coordonnées locales*. Dans ce cas, l'accès aux éléments se fait indépendamment de leur position réelle dans le tableau.

L'interface des tableaux à N dimensions est constituée de plusieurs méthodes que doit fournir le programmeur :

- `lget` et `lset` renseignent FT-GReLoSSS sur comment accéder aux éléments du tableau local en utilisant les coordonnées locales ;
- `data_possessed` et `data_needed` définissent le partitionnement des données parmi les processeurs ;
- `swap` échange le contenu de deux tableaux à N dimensions. Cette méthode est utilisée au sein des squelettes proposés par FT-GReLoSSS. La manière de réaliser l'échange (par échange de pointeur ou par recopie) est choisie par le programmeur ;
- `lgetAddr` retourne l'adresse mémoire d'un élément quelconque du tableau local ; elle est utilisée par FT-GReLoSSS pour réaliser un certain nombre d'optimisations.

L'interface des tableaux à N dimensions de FT-GReLoSSS fournit d'autres méthodes. Nous pouvons citer `gget` et `gset`. Ces dernières permettent d'accéder aux éléments du tableau local en utilisant les coordonnées globales. Ces méthodes sont définies à partir des méthodes d'accès `lget` et `lset` mais aussi à partir des méthodes de partitionnement `data_possessed` et `data_needed`.

5.6.2 Plan de routage

FT-GReLoSSS introduit le concept de plan de routage auquel est assignée la tâche de déterminer l'ensemble des communications à réaliser à une étape. Ceci laisse la possibilité de réaliser ensuite les communications de manière *efficace*.

À partir des méthodes de partitionnement définies par le programmeur au sein de son tableau distribué, FT-GReLoSSS déduit le plan de routage global. Pour un envoi d'un processus P_i à un processus P_j, FT-GReLoSSS constitue sur P_i un tampon mémoire dans lequel il recopie l'ensemble des données du tableau distribué à envoyer à P_j. Sur ce dernier, FT-GReLoSSS constitue un tampon de réception adapté. Dès que la réception est achevée, FT-GReLoSSS recopie les données reçues aux bons endroits dans le tableau distribué. Dans le cadre de tableaux à N dimensions où les données à envoyer ne sont pas toutes contiguës, cette manière de procéder permet de réaliser un unique envoi et d'économiser en latence réseau. En revanche, elle apporte un léger surcoût lié aux recopies et demande de disposer d'un peu plus de mémoire vive (RAM).

Pour l'exécution du plan de routage, FT-GReLoSSS offre actuellement le choix entre deux stratégies. La première plannifie tous les envois et les réceptions en même temps tandis que la seconde utilise un schéma considérant une topologie en anneau des processus. Les communications se font selon le schéma de la figure 5.19 : chaque processus plannifie les communications avec ses voisins de gauche et de droite immédiats puis ses voisins de gauche et de droite à distance 2 et ainsi de suite jusqu'à la distance n. Les voisins immédiats sont ceux situés à une distance de 1. La seconde stratégie avait initialement été implantée dans le cadre de la distribution d'un problème mettant en jeu un tableau distribué à 1 dimension [28].

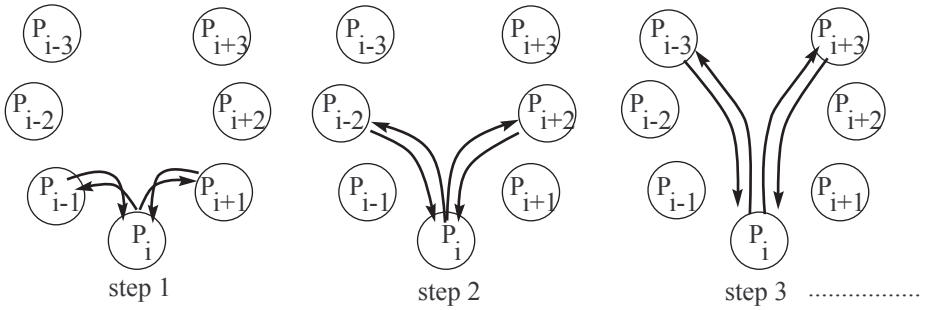


FIGURE 5.19 – Stratégie évoluée de plannification des communications sur un anneau de processus. Seuls les envois et les réceptions du processus i ont été représentés.

Les deux stratégies utilisent des primitives MPI asynchrones afin de permettre le recouvrement entre les communications. Les primitives utilisées sont `MPI_Issend` et `MPI_Irecv`. L'emploi des primitives asynchrones `MPI_Isend` et `MPI_Ibsend` a été envisagé pour les envois. Ces dernières n'ont cependant pas été retenues. En effet, `MPI_Isend` aurait permis d'éviter la synchronisation supplémentaire issue du rendez-vous réalisé par `MPI_Issend`. Cependant, `MPI_Isend` a été écarté car son implantation n'est pas standardisée [89]. Elle peut donc avoir un comportement différent voire imprévisible d'une implantation MPI à l'autre ce qui n'est pas souhaitable. Quant à `MPI_Ibsend`, son implantation est standardisée mais nécessite l'allocation de tampons mémoire supplémentaires alors qu'on exploite déjà un tampon explicite où l'on rassemble les données à envoyer. Cette allocation induirait une augmentation de la mémoire consommée. Enfin, d'après des mesures que nous avons réalisées, dans le cadre de la distribution d'une application de contrôle

stochastique [28], nous avons trouvé que le gain apporté par `MPI_Ibsend` était marginal. La consommation mémoire plus faible et les performances semblables nous ont conduits à retenir `MPI_Issend` au lieu de `MPI_Ibsend`.

Actuellement, le plan de routage est optimisé pour des tableaux à N dimensions admettant un ordre de stockage contigu selon la dimension la plus élevée (ou ordre de stockage des tableaux en langage C). Le plan de routage peut être amélioré en tenant compte d'ordres de stockage arbitraires. Cette fonctionnalité devra être ajoutée au niveau de la classe `Domain`. Nous envisageons un système similaire à celui mis en place pour les tableaux *Array* de la bibliothèque *Blitz++* [13] ou les tableaux *multiarray* de la bibliothèque Boost [14]. Ce système permet de spécifier des ordres de stockage arbitraires.

Enfin, FT-GReLoSSS intègre un critère pour différencier automatiquement le type du plan de communication. Il peut s'agir d'un échange de frontières ou d'une circulation de données. Selon le cas, le rôle de la deuxième structure de données (cf. Section 5.3.2) change. Dans le premier cas, elle est en lecture et en écriture comme pour l'algorithme de relaxation (cf. Figure 5.14). Dans le second cas (cf. Figure 5.6), la première structure de données est en lecture tandis que la seconde est en écriture. Pour le plan de routage cela pose un problème dans la mesure où l'endroit de lecture des données, qui est utilisé pour la réalisation des envois, varie d'une application à l'autre. FT-GReLoSSS s'en sort en déterminant le type de l'application à partir des méthodes de partionnement fournies par le programmeur. Le critère utilisé est le suivant : un schéma de communication correspond à un schéma de circulation si et seulement si pour toutes les phases de communication, les données possédées et données nécessaires ne se recouvrent pas.

5.7 Résumé du framework FT-GReLoSSS

Dans ce chapitre, nous avons présenté le framemork FT-GReLoSSS qui résulte de l'application des principes du modèle MoLOToF au paradigme de programmation parallèle SPMD. L'application de MoLOToF a abouti à deux squelettes tolérants aux pannes qui facilitent l'implantation d'applications SPMD à échange de frontières mais également à circulation de données. Pour ce faire, le programmeur doit essentiellement s'occuper de fournir la fonction de calcul et les fonctions de partitionnement. Les définitions des fonctions de partitionnement suffisent pour définir l'ensemble des communications qui doivent avoir lieu entre les différents processus de l'application. Le programmeur n'a donc pas à manipuler directement des primitives d'envoi et de réception : elles sont prises en charge par le plan de routage. Cette approche permet aussi de profiter de différents schémas de communications prédéfinis dans FT-GReLoSSS.

Par ailleurs, les squelettes facilitent l'ajout d'une tolérance aux pannes efficace en intégrant d'une part un protocole efficace de réalisation de points de reprise : l'absence de messages de coordination rend le protocole très adapté pour le passage à l'échelle. L'efficacité est d'autre part atteinte en impliquant le programmeur dans la constitution du contenu des points de reprise de manière à obtenir des tailles de points de reprise réduites. La réalisation des points de reprise s'effectue selon une fréquence configurable par le programmeur.

Enfin, les squelettes permettent de proposer un protocole original de niveau applicatif pour coordonner les actions des processus applicatifs. Ce protocole rend effectif le fonctionnement piloté d'applications implantées avec FT-GReLoSSS. Dans le cas présent, il est prévu pour déclencher des points de reprise depuis l'environnement mais pourrait être étendu pour permettre de changer la période de réalisation des points de reprise.

Chapitre 6

ToMaWork : un nouveau framework Maître-Travailleur tolérant aux pannes

ToMaWork (Tolerant Master-Worker) est un nouveau framework *Maître-Travailleur* (ou *Master-Worker* (MW)) tolérant aux pannes qui résulte de l'application des principes du modèle MoLO-ToF au paradigme de parallélisation Maître-Travailleur.

Après une présentation du type d'applications Maître-Travailleur considerées (Section 6.1), nous nous intéressons à l'architecture originale de ToMaWork (Section 6.2) qui comprend un modèle de parallélisation et des squelettes tolérants aux pannes, au-dessus d'un système à mémoire partagée virtuelle. Ensuite nous illustrons la simplicité d'écriture d'une application Maître-Travailleur avec ToMaWork (Section 6.3). Enfin, nous détaillons la solution de tolérance aux pannes adoptée par ToMaWork (Section 6.4) qui mêle une tolérance aux pannes dite « standard » avec une collaboration entre le programmeur et le framework. La tolérance aux pannes « standard » mêle elle-même des mécanismes de niveau framework avec des fonctionnalités fournies par l'intergiciel sous-jacent.

6.1 Types de programmes *Maître-Travailleur* considérés

Le paradigme de parallélisation *Maître-Travailleur* [27] met en jeu deux types d'entités : un *processus maître* et plusieurs *processus travailleurs* que nous appelerons dorénavant simplement maître et travailleurs. Le principe de ce paradigme est que le maître divise le problème initial à résoudre (ou tâche initiale) en N sous-problèmes *indépendants* (ou sous-tâches) qui sont distribués aux travailleurs. Les solutions partielles (ou résultats) produites par les travailleurs sont retournées au maître qui les utilise pour produire le résultat final. Les communications dans ce paradigme sont limitées aux échanges entre le maître et les travailleurs : il n'y a pas de communications entre les travailleurs.

Il existe deux variantes principales du paradigme Maître-Travailleur en fonction de la manière dont sont distribuées les sous-tâches aux travailleurs [27] : la première est le *Maître-Travailleur avec équilibrage statique* et la seconde est le *Maître-Travailleur avec équilibrage dynamique*. Dans le Maître-Travailleur statique, toutes les sous-tâches sont distribuées parmi les travailleurs disponibles au tout début de l'exécution de l'application. Ceci est accompli en partageant la tâche initiale en autant de sous-tâches qu'il y a de travailleurs ou en distribuant *équitablement* les sous-tâches issues du partage parmi les travailleurs. Le maître collecte ensuite les résultats et produit le résultat final. Dans le Maître-Travailleur dynamique, tous les travailleurs reçoivent une (sous-)tâche au début de l'application. Aussitôt qu'ils retournent un résultat, ils reçoivent

une nouvelle tâche (s'il en reste) prise dans la réserve de tâches. Ce schéma se poursuit jusqu'à ce qu'une condition d'arrêt satisfaisante soit atteinte. Une condition possible est l'épuisement des tâches. Dans ce type de Maître-Travailleur, nous pouvons distinguer deux variantes classiques : dans la première, les tâches traitées sont uniquement celles qui ont été définies au début de l'application tandis que dans la deuxième variante, de nouvelles tâches peuvent être générées (1) par le maître suite à la réception de résultats ou (2) par les travailleurs au cours du traitement de leurs tâches. Cette deuxième forme – où les travailleurs génèrent des tâches – n'est pas supportée par la solution de tolérance aux pannes que nous introduisons (cf. Section 6.4). Une troisième variante consiste à adopter un mécanisme de *vol de cycles* (ou *work stealing*) au paradigme Maître-Travailleur, pour réaliser un équilibrage de charge plus efficace. Ce mécanisme se rencontre de plus en plus fréquemment dans des applications pour processeurs multi-cœurs. *A priori*, il semble tout à fait compatible avec notre modèle de tolérance aux pannes (MoLOToF) et le framework que nous avons développé (ToMaWork), mais nous n'avons pas eu le temps de pousser ces investigations dans le cadre de cette thèse.

Dans toutes les variantes du Maître-Travailleur dynamique, plusieurs échanges sont effectués entre le maître et les travailleurs (contre un seulement pour le Maître-Travailleur statique). Les processeurs les plus rapides achèvent leurs tâches plus rapidement et se voient donc attribuer de nouvelles tâches plus fréquemment. Cette manière de procéder possède un surcoût de communication et de synchronisation plus important, mais permet de réaliser un équilibrage de charge dynamique.

6.2 Architecture de ToMaWork

Outre les entités de maître et de travailleur, nous pouvons dégager le concept de *réservoir* (*pool*) de tâches et de résultats. Ce concept permet de structurer davantage le paradigme Maître-Travailleur entre (1) composants spécialisés dans la gestion des données (le réservoir), et (2) composants spécialisés en calcul (le maître et les travailleurs). Sachant qu'aucun couplage fort entre les entités n'est requis, les échanges d'information peuvent très bien avoir lieu au travers du tiers représenté par le réservoir. Un avantage de cette solution est de migrer une partie de la complexité du maître et des travailleurs sur un tiers. Au-delà du bénéfice au niveau de la conception logicielle, cette approche permet d'une part d'utiliser des couches intergicielles existantes pour mettre en œuvre le tiers et d'autre part, de bénéficier de méchanismes de tolérance aux pannes opérationnels fournis par ces couches.

La technologie JavaSpaces, dont l'origine remonte au modèle de programmation Linda [56], permet à des applications d'échanger des objets `Java` à travers une mémoire partagée virtuelle que l'on appelle également *Javaspace* ou *espace de tuples*. Cette mémoire fonctionne de manière associative sur la base d'un motif correspondant à la caractéristique de l'objet (ou des objets) recherché(s). La technologie JavaSpaces fournit une API pour intéragir avec un Javaspace. Cette API permet de déposer un objet dans un Javaspace (opération `write`), d'en retirer une copie (opération `read`), ou tout simplement d'en retirer l'objet lui-même (opération `take`). Cette API est simple, mais suffisamment expressive pour permettre de développer rapidement et facilement une grande variété d'applications distribuées. Notamment celles mettant en jeu des communications complexes et non entièrement régulières.

En utilisant la technologie JavaSpaces pour mettre en œuvre le réservoir, nous avons abouti, pour notre framework ToMaWork, à l'architecture illustrée à la figure 6.1. ToMaWork abstrait les entités *maître* et *travailleur* ainsi que les données *tâche* et *résultat* que le maître et les travailleurs s'échangent via l'*espace de tuples* (Javaspace). À partir des abstractions `Master`, `Worker`,

`Task` et `Result` qui en résultent (cf. Figure 6.1), le programmeur dérive sa propre application Maître-Travailleur constituée de `User_Master`, `User_Worker`, `User_Task` et `User_Result`. Cette application bénéficie de la solution de tolérance aux pannes de ToMaWork dont les éléments constitutifs sont représentés en traits pointillés (couleur bleue) sur la figure 6.1.

Les relations entretenues dans ToMaWork entre les différentes entités en absence de tolérance aux pannes constituent ce que nous appelons le *modèle de parallélisation* de ToMaWork (Section 6.2.1) qui est incarné dans des squelettes (Section 6.2.2). La solution de tolérance aux pannes intégrée dans ces squelettes est développée ultérieurement (Section 6.4).

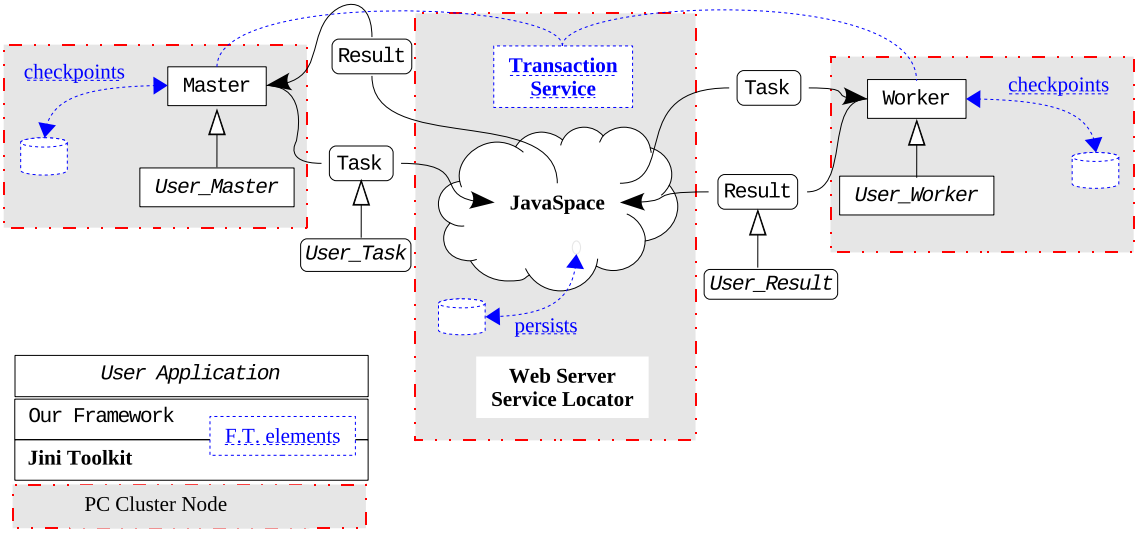


FIGURE 6.1 – Vue d'ensemble de l'architecture du framework ToMaWork.

6.2.1 Modèle de parallélisation de ToMaWork

Nous avons vu à la section 6.1 l'existence de différentes variantes du paradigme Maître-Travailleur. Le modèle de parallélisation de ToMaWork définit de manière plus précise la séquence des actions réalisées par les entités maître et travailleur, et ceci dans le cadre de l'architecture particulière d'espace de tuples sur laquelle il repose. Le maître dépose l'ensemble des tâches initiales à destination des travailleurs, dans l'espace de tuples. Puis il se met à observer l'espace de tuples dans l'attente de résultats. Les travailleurs retirent des tâches de l'espace de tuples et y déposent des résultats. À chaque résultat récupéré, le maître applique éventuellement un post-traitement intermédiaire et évalue la condition d'arrêt du calcul distribué. Le post-traitement peut aboutir à la création de nouvelles tâches qui seront déposées dans l'espace de tuples. Lorsque la condition d'arrêt est vérifiée, le maître dépose dans l'espace de tuples un message spécial pour signaler la fin du calcul aux travailleurs, et ainsi terminer proprement l'application. De son côté, la fonction d'un travailleur consiste donc à retirer une tâche de l'espace de tuples, la traiter puis à en déposer le résultat dans l'espace de tuples. Ceci continue jusqu'à la lecture de l'ordre d'arrêt dans l'espace de tuples. Afin de recouvrir une partie des temps de communication avec des traitements, le maître réalise (1) le dépôt des tâches ainsi que (2) le retrait et le traitement des résultats en utilisant deux processus distincts. Chaque travailleur est aussi constitué de deux processus : un pour retirer une tâche, la traiter et en déposer le résultat ; et l'autre pour scruter l'espace de tuples dans l'attente du message issu du maître et qui marque la fin de l'application.

Ainsi le modèle de parallélisation de ToMaWork met en œuvre un Maître-Travailleur dyna-

mique avec possibilité de créer de nouvelles tâches suite à la réception d'un résultat. La généralité d'un Maître-Travailleur dynamique permet de simuler un Maître-Travailleur statique : ceci s'accomplit simplement en générant un nombre de tâches initial égal au nombre de travailleurs, et en ne générant pas de tâches supplémentaires à la réception d'un résultat. Le modèle décrit ici aboutit aux deux squelettes proposés par ToMaWork, et font l'objet de la prochaine section.

6.2.2 Squelettes proposés par ToMaWork

ToMaWork propose deux squelettes : un pour l'entité maître et l'autre pour l'entité travailleur du paradigme Maître-Travailleur. Une version simplifiée de ces squelettes est présente dans les figures 6.2 et 6.3. Notamment, ces squelettes sont exposés en supposant que toutes les phases sont réalisées par un processus unique. Dans la pratique, deux processus sont mis en œuvre pour réaliser certaines phases de manière concurrente (cf. Section 6.2.1), mais leur présence aurait nuit à la clarté.

```
1  MW_Skel_master()
2  {
3     JavaSpace js;                                              (0)
4     Queues TaskQ, ResQ;
5
6     /* INITIALISATIONS */                                      (1)
7     // Découverte du Javaspace
8     // ...
9
10    /* DISTRIBUTION INITIALE DES TACHES */                     (2)
11    for (i=0; i<N; i++)
12       // Écriture d'une tâche dans le Javaspace
13       write_task(js, TaskQ.get(i));
14
15    /* COLLECTION DES RESULTATS */                             (3)
16    boolean end = false;
17    while (!end)
18    {
19       // Retrait d'un résultat du Javaspace et
20       // stockage dans la file ResQ
21       take_res(js, ResQ);
22
23       // Evaluation de la condition d'arrêt
24       end = checkTermination(TaskQ, ResQ);
25
26       /* SOUMISSION EVENTUELLE DE NOUVELLES TACHES */        (4)
27       /*                    OU                          */
28       /*  DEMANDE DE TERMINAISON DE L'APPLICATION       */
29       if (!end)
30          // Écriture des nouvelles tâches dans le Javaspace
31          while (!empty(TaskQ))
32             write_task(js, TaskQ.getNext());
33       else
34          // Ecriture du message d'arrêt des calculs
35          write_stop_order(js);
36    }
37
38    /* POST-TRAITEMENT FINAL DES RESULTATS */                  (5)
39    post_process(ResQ);
40 }
```

FIGURE 6.2 – Squelette simplifié du processus maître fourni par ToMaWork.

Le squelette du maître (cf. Figure 6.2) considère une réference vers l'espace de tuples (l. 3)

ainsi que deux files `TaskQ` et `ResQ` (l. 4) dans lesquelles sont placés respectivement l'ensemble des tâches à distribuer et l'ensemble des résultats reçus. Le squelette est décomposé en plusieurs phases. La *phase d'initialisation* (l. 6 à 8) comporte notamment la découverte de l'espace de tuples et l'initialisation des files. En particulier, l'ensemble des tâches initiales à distribuer sont placées dans la file `TaskQ`. Pendant la *phase de distribution* qui s'ensuit (l. 10 à 13), les tâches sont retirées de la file `TaskQ` pour être déposées dans l'espace de tuples (*i.e.* : le maître ne conserve pas de copies des tâches). À la fin de la distribution initiale des tâches, le maître entre dans la phase de *collecte des résultats* (l. 15 à 36). Durant cette phase, le maître retire un résultat (l. 21) et réévalue la condition d'arrêt (méthode `checkTermination`, l. 24). Selon le résultat de l'évaluation, il procède à l'écriture de nouvelles tâches qui ont peut-être été créées pendant l'évaluation de la condition d'arrêt ou bien il écrit l'ordre d'arrêt dans l'espace de tuples. En pratique, le dépôt de tâches de cette étape est réalisé de manière concurrente aux autres opérations décrites dans la phase de collection des résultats. Pour finir, le maître procède au post-traitement de l'ensemble des résultats (l. 39).

```
1  MW_Skel_worker()
2  {
3      JavaSpace js;                                              (0)
4
5      /* INITIALISATIONS */                                      (1)
6      // Découverte du Javaspace
7      // ...
8
9      boolean end = false;
10     while (!end) {
11         /* RETRAIT D'UNE TACHE */                              (2)
12         Task t = take_task(js);
13
14         /* TRAITEMENT DE LA TACHE ET ENVOI DU RESULTAT */      (3)
15         // Calcul du résultat
16         Result r = compute(task);
17         // Écriture du résultat obtenu dans le Javaspace
18         write_res(js, r);
19
20         /* VERIFICATION DE LA PRESENCE D'UN ORDRE d'ARRÊT */   (4)
21         if (read_stop_order(js))
22             end = true;
23     }
24 }
```

FIGURE 6.3 – Squelette simplifié du processus travailleur fourni par ToMaWork.

Le squelette d'un travailleur (cf. Figure 6.3) est plus simple et consiste en une série d'initialisations (l. 5 à 7) qui comprend — comme pour le maître — la découverte de l'espace de tuples. Ces initialisations sont suivies de la phase principale (l. 10 à 23) qui est constituée d'une boucle pendant laquelle le travailleur récupère une tâche (l. 12), la traite (l. 16) et en dépose le résultat (l. 18) dans l'espace de tuples. Avant de recommencer ce cheminement, le travailleur vérifie (l. 21 à 22) si un ordre d'arrêt est présent dans l'espace de tuple auquel cas il met fin à son exécution. En pratique, cette vérification est réalisée de manière concurrente aux autres phases décrites précédemment. Ainsi, le travailleur peut s'arrêter aussitôt qu'il reçoit le message sans avoir à attendre la fin du traitement de la tâche en cours.

6.3 Étapes de développement avec le framework ToMaWork

Dans cette section nous montrons comment utiliser notre framework ToMaWork pour écrire une application de calcul distribué selon le paradigme Maître-Travailleur. Nous présentons dans un premier temps l'interface proposée au programmeur par ToMaWork (Section 6.3.1) que nous illustrons, dans un second temps, à travers la mise en œuvre d'une application distribuée du calcul de la constante π par une méthode de Monte Carlo classique (Section 6.3.2).

6.3.1 Interface pour la programmation d'applications sous ToMaWork

ToMaWork fournit une API pour le développement d'applications selon le paradigme de parallélisation Maître-Travailleur. Cette API est constituée de quatre classes abstraites : `Master`, `Worker`, `Task` et `Result` dont le programmeur hérite pour définir les classes du maître, des travailleurs, des tâches et des résultats de son application Maître-Travailleur. Selon la classe mère, le programmeur peut être amené à définir dans la classe fille résultante, d'éventuelles méthodes abstraites qui ont été héritées (de la classe mère).

6.3.1.1 Implantation du maître

Pour le maître de l'application, le programmeur doit d'abord hériter de la classe mère `Master` de ToMaWork. Dans la classe fille qui en résulte, il doit ensuite implanter les méthodes :

- `boolean checkTermination(Vector<T> tasks, Vector<R> results)`
- `void postProcess(Vector<R> results)`

La méthode `checkTermination` détermine s'il faut s'arrêter. Cette méthode est exécutée automatiquement à chaque retrait d'un résultat (cf. Section 6.2.2). La manière d'implanter la méthode `checkTermination` détermine le type de Maître-Travailleur : statique ou dynamique. En effet, pour un Maître-Travailleur statique cette méthode revient simplement à tester si nous avons récupéré autant de résultats que de tâches soumises. Pour un Maître-Travailleur dynamique, cela peut consister en une condition plus complexe telle que, par exemple, un critère de convergence. Cette méthode donne accès au programmeur aux listes de tâches et de résultats du maître.

La méthode `postProcess` permet de réaliser un post-traitement sur les résultats, et elle est appelée à la fin de l'application (cf. Section 6.2.2).

6.3.1.2 Implantation d'un travailleur

Pour le travailleur, le programmeur doit d'abord hériter de la classe mère `Worker`, puis dans la classe fille qui en résulte, il doit implanter la méthode :

- `R compute(T t)`

La méthode `compute` réalise le traitement associé à la tâche `t` qui lui est passée en paramètre et renvoie un résultat de type `R`.

Remarque : Aussi bien pour le maître que pour le travailleur, l'initialisation du framework est réalisée lors de l'appel du constructeur de la classe mère.

6.3.1.3 Implantation d'une tâche et d'un résultat

Pour implanter une tâche et un résultat, le programmeur doit hériter des classes correspondantes `Task` et `Result` fournies par ToMaWork. Cependant, contrairement aux classes `Master` et

`Worker`, les classes `Task` et `Result` n'imposent rien au programmeur (*i.e.* : pas de méthodes abstraites à définir). En revanche, `Task` et `Result` sont des entrées du Javaspace, et à ce titre, elles sont soumises à un certain nombre de restrictions imposées par la technologie JavaSpaces et auxquelles n'échappent pas les classes qui en héritent. Ainsi, toute nouvelle classe définie par le programmeur et qui hérite de `Task` ou de `Result` doit être sérialisable (au sens Java du terme), seuls les champs déclarés publics seront sérialisés, et enfin, la classe fille doit être munie d'un constructeur par défaut vide. La présence d'un tel constructeur découle de l'emploi des « *generics* » de Java, que nous utilisons pour obtenir généricité et sûreté de typage (*type safety*) (cf. [57] pour plus d'explications).

En pratique, nous conseillons au programmeur de ne mettre dans sa classe représentant une tâche que les données relatives à son traitement ; et de n'inclure dans sa classe représentant un résultat que le résultat. En particulier, le code de traitement des tâches, qui est identique à toutes les tâches, ne devrait être présent que sur les travailleurs.

Ces conseils sont importants à prendre en compte pour des raisons de performances de l'application et de la solution de tolérance aux pannes. À defaut de pouvoir le faire lorsque, par exemple, certaines données sont trop volumineuses, il est préferable d'inclure les informations permettant de récupérer ces données directement (*i.e.* : sans les faire transiter par le Javaspace).

Enfin, un autre intérêt de cette organisation est de permettre de déporter des objets non-sérialisables dans les travailleurs (*e.g.* : threads, flots, ...).

La section suivante illustre l'utilisation de l'API décrite précédemment pour implanter une application distribuée approximant la constante π par une méthode de Monte Carlo.

6.3.2 Pi : un exemple de calcul distribué de π avec une méthode de Monte Carlo

L'application « *Pi* » distribue le calcul de l'approximation de la constante π selon une méthode de Monte Carlo. Les méthodes de Monte Carlo sont très répandues dans de nombreux domaines allant de la physique nucléaire à l'économie (ou la finance). La manière d'appliquer une méthode de Monte Carlo varie d'un domaine à un autre mais revient dans tous les cas à utiliser des nombres aléatoires pour résoudre statistiquement un problème.

6.3.2.1 Description de l'algorithme

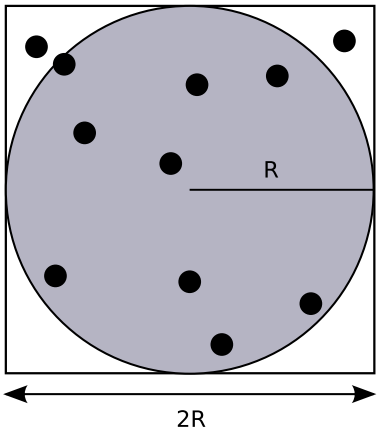


FIGURE 6.4 – Illustration du principe de calcul de π avec une méthode de Monte Carlo.

L'algorithme, qui est illustré à la figure 6.4, consiste à effectuer un certain nombre de tirages aléatoires *nb_tir* de points dans un espace 2D. Cet espace est délimité par un carré de côté $2R$ dans lequel est inscrit un disque de rayon R. Le décompte du nombre de points *nb_in* ayant atterri dans la surface du disque (zone grisée) permet d'approximer la valeur de π d'après la formule :

$$\pi = 4 \times \frac{nb\_in}{nb\_tir} \tag{6.1}$$

Une distribution possible de cet algorithme selon le paradigme Maître-Travailleur consiste à faire générer par le maître *nb_tir* points aléatoires du plan qu'il poste dans le Javaspace. Chaque travailleur prend un point, calcule sa position dans l'espace 2D et retourne un résultat booléen indiquant si le point se situe à l'intérieur du disque (valeur vraie) ou à l'extérieur (valeur fausse).

6.3.2.2 Mise en œuvre avec ToMaWork

Dans la version distribuée implantée avec ToMaWork, une tâche `PiTask` (cf. Figure 6.5a) correspond à un tirage aléatoire (*i.e.* : deux variables réelles `x` et `y`) et le résultat associé `PiResult` (cf. Figure 6.5b) se réduit à une variable booléenne (`isInside`) indiquant si le tirage correspondant se trouve dans le disque ou non. `PiTask` et `PiResult` héritent respectivement des classes `Task` et `Result` présentes dans le paquetage `mawork`[14] de ToMaWork.

```
1  public class PiTask
2        extends mawork.Task
3  {
4     public Double x, y;
5
6     public PiTask() {}
7
8     public PiTask(Double x, Double y)
9     {
10       this.x = x;
11       this.y = y;
12    }
13 }
```

(a) Code source de PiTask.

```
1  public class PiResult
2        extends mawork.Result
3  {
4     public Boolean isInside;
5
6     public PiResult() {}
7
8
9
10
11
12
13 }
```

(b) Code source de PiResult.

FIGURE 6.5 – Définitions des classes *PiTask* et *PiResult*.

Le calcul d'un travailleur consiste simplement à effectuer le test d'appartenance au disque. Ce test est contenu dans la méthode `PiWorker::compute` (cf. Figure 6.6, l. 9 à 21). Le constructeur (l. 4 à 7) effectue simplement un appel au constructeur de la classe mère en lui précisant le type des tâches (`PiTask.class`). Les deux arguments qui précèdent le type de tâches, et que nous avons omis, sont utilisés pour la découverte du Javaspace.

Alors que le code du travailleur est identique pour un Maître-Travailleur statique et dynamique, pour le maître cela conduit à deux implantations différentes au niveau des méthodes `checkTermination` et `postProcess` de la classe `PiMaster`. Dans la version statique (cf. Figure 6.7), la condition d'arrêt (l. 14 à 19) est vérifiée lorsque le nombre de résultats récupérés correspond au nombre de tâches distribuées. Le calcul de la valeur finale de l'approximation de π s'effectue dans la méthode de post-traitement `postProcess` (l. 22 à 33).

14. Pour « master-worker »

```
1  public class PiWorker extends mawork.Worker<PiTask, PiResult>
2  {
3     // Constructeur
4     public PiWorker(/*...*/, /*...*/)
5     {
6        super(/*...*/, /*...*/, PiTask.class);
7     }
8
9     // Méthode définissant le calcul à effectuer par les travailleurs
10    public PiResult compute(PiTask t)
11    {
12       double distance = (t.x * t.x) + (t.y * t.y);
13
14       PiResult result = new PiResult();
15
16       if (distance < 1)   result.isInside = false;
17       else                result.isInside = true;
18
19       return result;
20    }
21 }
```

FIGURE 6.6 – Maître-Travailleur statique/dynamique : code source du travailleur PiWorker.

```
1  public class PiMaster                          // VERSION STATIQUE DU MAÎTRE
2     extends mawork.Master<PiTask, PiResult>
3  {
4     private int nbPostedTasks; // Nombre total de tâches à déposer
5
6     // Constructeur: initialisation avec la liste des tâches initiales
7     public PiMaster(/*...*/, /*...*/, Vector<PiTask> initialTasks)
8     {
9        super(/*...*/, /*...*/, initialTasks, PiResult.class);
10       nbPostedTasks = initialTasks.size();
11    }
12
13    // Méthode de détection de la terminaison de l'application
14    public boolean checkTermination(Vector<PiTask>   tasks,
15                                    Vector<PiResult> results)
16    {
17       // Condition d'arrêt de l'application
18       return results.size() == nbPostedTasks;
19    }
20
21    // Méthode de post traitement des résultats
22    public void postProcess(Vector<PiResult> results)
23    {
24       // Calcul de pi d'après l'ensemble des résultats obtenus
25       int nbIn = 0, nbTir = 0;
26       for (int i = 0; i < results.size(); i++) {
27          if (results.get(i).isInside)
28             nbIn++;
29
30           nbTir++;
31       }
32       System.out.println("pi = " + (double)((double)4*nbIn/nbTir));
33    }
34 }
```

FIGURE 6.7 – Maître-Travailleur statique : code source du maître PiMaster statique.

Dans la version dynamique (cf. Figure 6.8), un post-traitement partiel est effectué dans la méthode d'évaluation de la condition d'arrêt `checkTermination`. Il consiste à calculer l'approximation courante de π (l. 17 à 22) selon la formule 6.1. Des tirages aléatoires supplémentaires (ou tâches) sont générés si l'approximation n'est pas satisfaisante (l. 24 à 32). Le code présent aux lignes 27 à 30 génère un nombre de tâches additionnelles deux fois plus important que le nombre de travailleurs de manière à pouvoir tous les approvisionner et ainsi limiter le phénomène de famine.

```
1  public class PiMaster                                  // VERSION DYNAMIQUE DU MAÎTRE
2     extends mawork.Master<PiTask, PiResult>
3  {
4     // Variables de décompte des points à l'intérieur et à l'extérieur du disque
5     int nbIn = 0, nbTir = 0;
6
7     // Constructeur: initialisation avec la liste des tâches initiales
8     public PiMaster(/*...*/, /*...*/, Vector<PiTask> initialTasks)
9     {
10       super(/*...*/, /*...*/, initialTasks, PiResult.class);
11    }
12
13    // Méthode de détection de la terminaison de l'application
14    public boolean checkTermination(Vector<PiTask>   tasks,
15                                    Vector<PiResult> results)
16    {
17       if (results.lastElement().isInside)
18          nbIn++;
19
20       nbTir++;
21
22       double piApprox = (double)((double)4*nbIn/nbTir);
23
24       boolean enough = (Math.abs(Math.PI-piApprox) < 0.5);
25
26       if (!enough && (tasks.size() == 0)) {
27          for (int i = 0; i < 2*getNumWorkers(); ++i) {
28            boolean x = Math.random();
29            boolean y = Math.random();
30            tasks.addTask(new PiTask(x, y));
31          }
32       }
33
34       return enough;
35    }
36
37    // Méthode de post traitement des résultats
38    public void postProcess(Vector<PiResult> results)
39    {
40       System.out.println("pi = " + (double)((double)4*nbIn/nbTir);
41    }
42 }
```

FIGURE 6.8 – Maître-Travailleur dynamique : code source du maître PiMaster dynamique.

En ce qui concerne la génération initiale des tâches, elle est réalisée entièrement par le programmeur qui les transmet au maître au moment de son instanciation. En effet, que ce soit pour le maître statique ou dynamique, le constructeur de la classe mère `Master` reçoit en paramètre une liste de tâches sous forme d'un vecteur de tâches (cf. Figure 6.8, lignes 7 à 11 et Figure 6.8, lignes 8 à 11)

Le lecteur attentif aura remarqué que le critère de convergence (cf. Figure 6.8, ligne 24) utilise

la valeur approchée de π du paquetage `Math` de Java. Ceci est un choix pour le moins paradoxal vis-à-vis du but annoncé qui est l'approximation de π. C'est un choix que nous avions initialement privilégié pour pouvoir illustrer les Maître-Travailleur statique et dynamique selon une seule et même application de Monte Carlo qui soit très simple. En pratique, l'estimation de π utilise d'autres méthodes, plus rapides, comme celles fondées sur l'une des nombreuses expressions de π sous forme d'une série infinie [160]. À défaut d'avoir trouvé un autre critère de convergence palliant la faiblesse du premier, nous nous sommes tenus à notre choix initial et prions le lecteur de ne pas nous en tenir rigueur.

6.4 Tolérance aux pannes dans ToMaWork

Dans le paradigme Maître-Travailleur, l'entité maître occupe une place centrale. En effet, les seules communications existantes s'effectuent entre le maître et les travailleurs exclusivement. De par sa position, le maître peut naturellement maintenir des informations sur toute l'application de manière à faire face à des pannes. En cas de panne d'un travailleur, la connaissance du travailleur et de la tâche qui lui était assignée rend possible la redistribution de la tâche à un autre travailleur. Éventuellement le travailleur défaillant peut être relancé et reprendre de nouvelles tâches. En cas de panne du maître, toute l'application peut être relancée à partir d'une sauvegarde du maître lequel se charge alors de redistribuer adéquatement les tâches. Cette approche est celle adoptée traditionnellement dans ce type d'application. Mais il en résulte que de nombreuses activités se voient concentrées sur le maître.

Dans le cas de ToMaWork, le choix d'une architecture à mémoire partagée virtuelle permet d'une part de migrer la complexité liée aux communications sur l'espace de tuples. D'autre part, le maître n'est plus surchargé de la gestion de la tolérance aux pannes de toute l'application. En effet, dans la solution adoptée, chaque entité devient responsable de sa propre tolérance aux pannes et l'état de l'application devient distribué sur chacune d'entre elles (*i.e.* : il n'est plus concentré sur le maître). Enfin, la panne d'une entité n'entraîne pas l'interruption de toutes les autres pour effectuer la reprise sur panne : cette dernière est isolée sur la seule entité qui tombe en panne, et seule cette entité a donc besoin d'être redémarrée.

Cette tolérance aux pannes fait partie de l'offre dont bénéficie automatiquement toute application développée avec ToMaWork ; c'est la tolérance aux pannes que nous appelons « standard » ou de « série », et qui permet de faire face à des pannes temporaires (de machines ou de processus). La tolérance aux pannes standard peut être affinée et rendue plus efficace au travers d'une collaboration du programmeur avec le framework. Aux sections 6.4.1 et 6.4.2 nous détaillons ces deux types de tolérance aux pannes.

6.4.1 Tolérance aux pannes « standard » de ToMaWork

Dans cette section nous décrivons le protocole de tolérance aux pannes de ToMaWork qui se fonde partiellement sur des mécanismes de tolérance aux pannes offerts par l'environnement de l'espace de tuples.

6.4.1.1 Description du protocole de tolérance aux pannes de ToMaWork

Étant donné la distribution de l'état de l'application sur tous les processus qui participent, l'objectif de la tolérance aux pannes standard est de réussir à maintenir cohérents les états de tous les participants en présence de la panne d'un ou de plusieurs d'entre eux (pannes simultanées). Compte-tenu des interactions (ou dépendances) existant entre le maître, l'espace de tuples et les

travailleurs, cela revient à maintenir un état cohérent entre l'espace de tuples et l'ensemble des travailleurs d'une part, et puis d'autre part entre le maître et l'espace de tuples.

Lorsqu'un travailleur retire une tâche de l'espace de tuples, il la sauve localement et l'espace de tuples se « souvient » de cette tâche jusqu'à ce qu'il obtienne la confirmation du travailleur (qui l'a retirée) qu'il l'a bien enregistrée sur disque. Ainsi, si le travailleur tombe en panne avant de l'avoir sauvée, la tâche est toujours présente sur l'espace de tuples lequel peut la distribuer à un autre travailleur. Sinon, la tâche quitte de manière permanente l'espace de tuples mais il est garanti qu'elle est sauvée auprès d'un travailleur. De cette manière, même si le travailleur tombe en panne en traitant la tâche, cette dernière n'est pas perdue. Il peut donc en recommencer le traitement à son redémarrage. L'opération de « retrait-sauvegarde » où le travailleur retire une tâche et la sauve (localement) doit donc être réalisée de manière atomique.

De façon similaire, lorsqu'un travailleur écrit un résultat dans l'espace de tuples, il n'efface la sauvegarde de la tâche correspondante que lorsqu'il obtient la confirmation de l'espace de tuples que le résultat correspondant a été sauvé en lieu sûr. Ainsi, même si un problème survient lorsque le travailleur écrit le résultat sur l'espace de tuples, il sera en mesure de le refournir ; et si le travailleur tombe en panne après avoir transmis le résultat avec succès, mais avant d'avoir entamé une nouvelle tâche, il ne refera pas le traitement de la tâche dont il a déjà transmis le résultat. L'opération « d'écriture-effacement » où le travailleur écrit le résultat d'une tâche et efface la sauvegarde de cette dernière doit donc être réalisée de manière atomique.

Le maître maintient trois éléments : une liste de tâches à poster dans l'espace de tuples, un compteur du nombre de tâches actuellement postées dans l'espace de tuples, et une liste avec les résultats. À chaque fois que l'un de ces éléments change, le maître réalise un point de reprise de lui-même. Ce point de reprise comporte les trois éléments que nous venons de citer. Dans le cas général, ces trois éléments constituent l'information utile minimale permettant au maître de reprendre son exécution après être tombé en panne. En effet, il pourra écrire les tâches qu'il n'a pas pu écrire avant sa panne, il pourra traiter les résultats qu'il avait reçus avant sa panne avec les éventuels nouveaux résultats qu'il recevra. Par ailleurs, la liste de résultats ainsi que le nombre de résultats postés peuvent intervenir dans l'évaluation de la condition d'arrêt. Enfin, la liste de résultats est bien évidemment nécessaire au calcul du résultat final.

Cependant, à l'image des travailleurs, la simple réalisation d'un point de reprise ne suffira pas à maintenir la cohérence entre le maître et l'espace de tuples. Ainsi l'opération « d'écriture-sauvegarde » où le maître écrit une tâche dans l'espace de tuples et se sauvegarde doit donc être réalisée de manière atomique. Il en est de même pour l'opération de « retrait-sauvegarde » où le maître retire un résultat de l'espace de tuples et se sauvegarde.

Ainsi, alors que dans le premier cas décrit, l'atomicité porte sur la réalisation d'opérations par la couple « Travailleur – Espace de tuples », dans le second cas, elle porte sur la réalisation d'opérations par le couple le « Maître – Espace de tuples ».

6.4.1.2 Mise en œuvre du protocole au-dessus de mécanismes intergiciels bas-niveau de tolérance aux pannes

ToMaWork est écrit en Java et il est construit au-dessus de plusieurs services Jini, ce qui lui permet de bénéficier de mécanismes intergiciels existants, en particulier des mécanismes de tolérance aux pannes.

Premièrement, les opérations de réalisation de points de reprise et de reprise sur panne sont facilitées par le mécanisme de « sérialisation/désérialisation » intégré au langage Java. Ceci constitue un avantage intéressant surtout pour les programmeurs non expérimentés. D'autres

langages de programmation (*i.e.* : C/C++) ont recours à des bibliothèques tierces qui n'atteignent pas le même niveau d'automatisation.

Deuxièmement, le service Javaspace de Jini (appelé *outrigger*) peut être rendu persistant. Par défaut, les entrées sont simplement présentes en mémoire vive. En mode persistant, le Javaspace écrit sur disque toute entrée qui est écrite dans le Javaspace, et il efface du disque toute entrée qui en est retirée. Ce mécanisme assure qu'en cas de redémarrage, le Javaspace sera exactement rempli avec les mêmes entrées qu'il avait avant de tomber en panne. Bien entendu, ce mécanisme introduit un surcoût à l'exécution qui dépend de la technologie de stockage utilisée mais aussi de la taille des entrées.

Troisièmement, ToMaWork utilise le service de transactions de Jini (appelé *mahalo*) afin de permettre au protocole de tolérance aux pannes standard de ToMaWork, de réaliser certaines opérations de manière atomique. Pour réaliser de manière atomique ses opérations « d'écriture-sauvegarde » et de « retrait-sauvegarde » le maître intègre les opérations correspondantes au sein d'une transaction. Dans le cas de l'opération « d'écriture-sauvegarde » (cf Figure 6.9), le maître initialise une transaction (ligne 1) au sein de laquelle (lignes 2 à 4) il procède d'abord à l'écriture d'une tâche dans le Javaspace puis ensuite à la sauvegarde de son état. La transaction n'est validée que si ces deux opérations s'accomplissent avec succès. En cas d'échec de l'une d'entre elles ou de panne du maître, ces opérations doivent être défaites. Pour l'opération sur le Javaspace, c'est son invocation dans le cadre de la transaction qui provoque l'annulation automatique. En revanche, l'opération de sauvegarde du maître, doit être prise en charge par ToMaWork. S'il s'agit d'un problème d'écriture comme une absence de droits en écriture ou une insuffisance de place sur disque, des exceptions sont levées et capturées par ToMaWork. Ce dernier se charge alors d'effacer l'éventuel fichier de sauvegarde incomplet. Si c'est une panne du maître alors l'action entreprise par ToMaWork dépend du moment d'occurrence de la panne :

1. Pour les pannes intervenant avant le début de la réalisation de la sauvegarde (avant la ligne 4), ToMaWork n'a aucune opération à défaire. L'opération sur le Javaspace est prise en charge par le service de Javaspaces ; il suffit donc au maître de redémarrer à partir de sa dernière sauvegarde.
2. Pour les pannes intervenant au moment de l'opération de sauvegarde (ou d'effacement de sauvegarde), le fichier de sauvegarde est laissé dans un état corrompu qui sera détecté et effacé au moment de la reprise ; encore une fois, il suffit au maître de redémarrer à partir de son avant-dernière sauvegarde (car la dernière n'a pas abouti).
3. Enfin, pour les pannes intervenant après l'opération de sauvegarde mais avant l'effacement de l'avant-dernière sauvegarde, ToMaWork a besoin de déterminer si elles se sont produites avant ou après la validation de la transaction. En effet, une panne avant la validation signifie que l'opération sur le Javaspace a été défaite et que par conséquent le maître doit redémarrer à partir de son avant-dernière sauvegarde. Au contraire, pour une panne après la validation le maître doit redémarrer à partir de sa dernière sauvegarde. La difficulté provient du fait que, pour ces pannes, il existe deux sauvegardes saines du maître lesquelles ne diffèrent que par l'absence d'une tâche dans la plus récente. Le maître doit donc être en mesure de vérifier à son redémarrage si la tâche a été écrite ou pas dans le Javaspace. Ainsi dans l'état actuel, le protocole n'est pas en mesure de faire face à ces pannes qui surviennent entre le moment où il a terminé sa sauvegarde courante et le moment où il efface la sauvegarde précédente. Une solution – fondée sur l'écriture de messages de contrôle dans le Javaspace et ne remettant pas en cause l'architecture du framework – a été conçue et se trouve en cours de validation.

```
1  transaction.init();                  // Initialisation de la transaction
2      Task t = TaskQ.getNext();        // Retrait d'un tâche de la file de tâches
3      write_task(js, t);               // Écriture de la tâche dans le Javaspace
4      checkpoint_master();             // Sauvegarde du maître
5  transaction.commit();                // Validation de la transaction
6  delete_old_checkpoint();             // Effacement de l'ancienne sauvegarde
```

FIGURE 6.9 – Pseudo-code pour la réalisation atomique de l'opération « d'écriture-sauvegarde » par le maître.

L'opération actuelle de « retrait-sauvegarde » du maître (cf. Figure 6.10) ainsi que les opérations actuelles de « retrait-sauvegarde » (cf. Figure 6.11) et « d'écriture-effacement » (cf. Figure 6.12) des travailleurs sont soumises à des limitations similaires étant donné qu'aucun mécanisme supplémentaire n'a été mis en place pour y remédier.

Par exemple, dans le cas de l'opération de « retrait-sauvegarde » (cf. Figure 6.11), si une panne survient entre la sauvegarde locale de la tâche (ligne 3) et la validation de la transaction (ligne 4), alors la transaction expire, la tâche demeure dans l'espace de tuples et elle devient à nouveau disponible pour d'autres travailleurs. Ainsi, lors de sa reprise sur panne, le travailleur défaillant possède une sauvegarde de la tâche mais il lui est impossible de savoir s'il est tombé en panne avant ou après la validation de la transaction. En effet, la tâche peut être absente du Javaspace soit parce qu'elle a été rétirée par un autre travailleur, soit parce que la transaction du travailleur défaillant a été validée. D'où la nécessité de mettre en place un mécanisme pour lever cette ambiguité et permettre une reprise correcte (*i.e.* : pas de perte ni de duplication de tâche).

```
1  transaction.init();                  // Initialisation de la transaction
2      Result r = take_res(js);         // Retrait d'une tâche du Javaspace
3      ResQ.add(r)                      // Stockage dans la file de résultats
4      checkpoint_master();             // Sauvegarde du maître
5  transaction.commit();                // Validation de la transaction
6  delete_old_checkpoint();             // Effacement de l'ancienne sauvegarde
```

FIGURE 6.10 – Pseudo-code pour la réalisation atomique de l'opération de « retrait-sauvegarde » par le maître.

```
1  transaction.init();                  // Initialisation de la transaction
2      Task t = take_task(js);          // Retrait d'une tâche du Javaspace
3      checkpoint_task(t);              // Sauvegarde de la tâche
4  transaction.commit();                // Validation de la transaction
```

FIGURE 6.11 – Pseudo-code pour la réalisation atomique de l'opération « retrait-sauvegarde » par le travailleur.

Enfin, une dernière difficulté inhérente à l'utilisation de transactions est celle du choix de la durée du bail que l'on associe à chaque transaction. Le bail constitue un détecteur de pannes : lorsqu'il expire cela correspond à une situation anormale. Nous distinguons deux situations. Dans la première, les opérations réalisées sous transaction prennent davantage de temps que

```
1  transaction.init();          // Initialisation de la transaction
2      write_res(js, r);        // Écriture du résultat dans le Javaspace
3  transaction.commit();        // Validation de la transaction
4  delete_task_checkpoint();    // Effacement de la sauvegarde de la
5                               // tâche traitée
```

FIGURE 6.12 – Pseudo-code pour la réalisation atomique de l'opération « d'écriture-effacement » par le travailleur.

celui initialement imparti par le bail et elles sont donc interrompues. Par ailleurs, leurs non-aboutissements peuvent être interprétés comme des pannes. Dans la deuxième situation, il s'agit d'une panne mais du fait d'un bail trop long, elle est détectée tardivement. Ainsi, un bail trop long conduit à une détection de pannes très lente et inversement un bail trop court associé à des opérations de transaction trop longues peut conduire à de fausses alertes. Enfin, un bail infini est à exclure car en cas de pannes, la transaction demeure non clôturée et l'entrée demeure indisponible aux autres travailleurs.

La stratégie adoptée consiste à établir un bail d'une durée donnée (éventuellement par le programmeur) que l'on fait renouveller régulièrement par un processus local jusqu'à ce que les opérations sous transaction se terminent. Par exemple, lorsque le maître réalise le retrait d'un résultat sous transaction, il associe à cette dernière un bail initial qui est renouvelé jusqu'à ce que l'opération sous transaction soit réalisée. Ceci permet d'assurer que l'opération pourra être réalisée même si sa durée excède la durée du bail initial qui lui a été associé. Si la JVM du maître (ou le nœud sur lequel s'exécute le maître) tombe en panne, le processus local au nœud ne renouvellera pas le bail lequel expirera et provoquera l'annulation des opérations sous transaction. Ce dernier doit alors être relancé à partir de son dernier point de reprise. Conséquemment, le service de transactions est très sollicité. Pour accroître le temps entre deux renouvellements et ainsi relâcher la pression mise sur le service de transactions, il serait intéressant d'utiliser un algorithme similaire à l'algorithme d'évitement de congestion de TCP qui est un algorithme à *augmentation additive/retrait multiplicatif* : à chaque renouvellement, le bail pourrait être augmenté de 10 secondes (par exemple) et divisé par 2 en cas de panne. Un tel algorithme permetterait de mieux approcher la durée optimale du bail, et ainsi de réduire le nombre de renouvellements ; mais nous n'avons pas pu encore l'expérimenter.

6.4.1.3 Activation de la tolérance aux pannes et reprise sur panne

Pour pouvoir bénéficier de la tolérance aux pannes standard de ToMaWork, il suffit au programmeur de faire hériter ses classes maître et travailleur des classes `Master` et `Worker` du paquetage java `tomawork`[15] au lieu de `mawork`. Par exemple, pour l'application *Pi* décrite à la section 6.3.2, cela revient simplement à avoir les déclarations :

```
public class PiWorker extends tomawork.Worker<PiTask, PiResult> { /* ... */ }
public class PiMaster extends tomawork.Master<PiTask, PiResult> { /* ... */ }
```

au lieu de :

```
public class PiWorker extends mawork.Worker<PiTask, PiResult> { /* ... */ }
public class PiMaster extends mawork.Master<PiTask, PiResult> { /* ... */ }
```

15. Pour « tolerant mawork »

En héritant de la classe mère `tomawork.Worker`, le programmeur est amené à implanter une nouvelle méthode `compute` :

– `R compute(WorkerCheckpoint c)`

Cette méthode est appelée par ToMaWork pendant la reprise sur panne d'un travailleur pour pouvoir redémarrer à partir d'éventuels points de reprise intermédiaires que le programmeur aurait définis au sein sa méthode `compute(T t)` initiale [16]. L'utilisation de la nouvelle méthode `compute` est illustrée à la section 6.4.2.2 (p. 109). À moins de définir des points de reprise intermédiaires, il suffira simplement au programmeur de donner un corps vide à cette méthode. Quant aux classes utilisateur définissant les tâches et les résultats, aucun changement n'est à apporter : elles continuent d'hériter des classes `mawork.PiTask` et `mawork.PiResult`.

Si le maître (ou un travailleur) tombe en panne, il suffit à l'utilisateur de le redémarrer en utilisant, sur la ligne de commande, le paramètre « `rego` » au lieu du paramètre « `go` ». Ces opérations devraient devenir entièrement transparentes à l'utilisateur dès que les mécanismes de tolérance aux pannes auront été entièrement implantés. D'ici là, une manière simple d'automatiser la procédure est l'utilisation d'un processus de surveillance des JVM qui les relance lorsque leur exécution est interrompue. Si c'est le Javaspace qui tombe en panne, le maître et les travailleurs le détecteront et attendront que le Javaspace soit redémarré.

Le Javaspace est également redémarré manuellement. Cependant, comme c'est un service Jini, il peut être rendu « `activatable` », ce qui lui permet d'être redémarré automatiquement en cas de panne [108].

6.4.2 Collaboration entre le programmeur et ToMaWork

Lorsqu'un travailleur retire une tâche de l'espace de tuples, cette dernière est automatiquement sauvée avant que le travailleur ne commence à la traiter (cf. Section 6.4.1). En cas de panne du travailleur pendant le traitement, le travailleur est redémarré et reprend le traitement de la tâche depuis le début. Les calculs qui sont effectués jusqu'au moment de la panne sont donc perdus. Ce comportement peut être gênant lorsque (1) le traitement de la tâche est long, (2) les pannes sont fréquentes, et (3) l'application est soumise à des contraintes de temps. Pour y remédier, ToMaWork permet au programmeur d'utiliser sa connaissance de l'environnement d'exécution, mais aussi sa connaissance de la logique et des structures de données applicatives, pour définir le contenu et la période de réalisation de points de reprise intermédiaires durant le traitement des tâches.

6.4.2.1 Description des collaborations possibles entre programmeur et framework

Tout d'abord, le programmeur a la possibilité d'activer ou de désactiver les mécanismes de tolérance aux pannes indépendamment sur les entités maître, travailleur et Javaspace. Par exemple, si le service Javaspace est hébergé par une machine très fiable, le programmeur peut choisir d'utiliser la version non persistante du Javaspace où les entrées sont simplement maintenues en mémoire. Nous exploitons ainsi la connaissance qu'a le programmeur de ses machines pour réduire le surcoût lié à la tolérance aux pannes. Nous exploitons aussi sa connaissance de l'application. Par exemple, pour une application où le maître génère des tâches jusqu'à ce que qu'un critère de convergence soit atteint, la perte de tâches peut *a priori* ne pas être grave dans la mesure où elles seront remplacées (ou compensées) par la génération de nouvelles tâches. Il est donc envisageable de relâcher la tolérance aux pannes sur les travailleurs en désactivant la

16. Cette méthode initiale a été introduite à la section 6.3.1.2 (p. 98).

réalisation de points de reprise. Si la duplication de tâches ne pose pas de problème dans la correction de l'application, nous pouvons également désactiver le mécanisme de transactions sur les travailleurs. Un exemple est celui de l'application *Pi* (cf. Section 6.3.2, p. 99) pour laquelle nous pouvons envisager dans sa version dynamique de désactiver la réalisation de points de reprise sur les travailleurs.

Le programmeur peut segmenter le code de traitement d'une tâche en blocs dont le temps d'exécution n'est pas négligeable. Un point de reprise est réalisé à la fin de chaque bloc. Il revient donc au programmeur de définir la fréquence de réalisation de points de reprise. Par exemple, si le calcul d'une tâche est composé d'une boucle, il peut décider de réaliser un point de reprise toutes les 10 itérations. Si le calcul est composé d'une séquence de trois « grosses » méthodes, il peut décider de réaliser un point de reprise après chacune d'entre elles.

À la sortie d'un bloc, la plupart de ses variables internes ne sont plus intéressantes, et se rappeler uniquement du résultat intermédiaire est suffisant pour poursuivre avec le prochain bloc. Ainsi, au lieu de sauver l'ensemble du contexte du processus, nous mettons à contribution la connaissance du programmeur en lui demandant de ne spécifier que le minimum de données nécessaires. Juste avant d'appeler la méthode de réalisation de point de reprise, le programmeur construit un objet sérialisable contenant toutes les informations nécessaires pour reprendre l'exécution à partir de ce point. Par exemple, le point de reprise pourrait contenir la valeur d'un accumulateur et l'indice de boucle dans le cas où le point de reprise est défini au sein d'une boucle. ToMaWork se charge d'écrire le point de reprise sur disque et d'enlever les précédents.

À la reprise, ToMaWork se charge de récupérer le point de reprise et d'en transmettre les informations à la méthode de calcul. Après avoir défini des points de reprise intermédiaires dans le code de calcul du travailleur, le programmeur doit spécifier la manière de reprendre les calculs à partir des informations sauvées. Pour faciliter la gestion des points de reprise, les positions des points de reprises sont manuellement numérotées selon un indice commençant à 1 (0 est reservé pour le point de reprise initial de la tâche qui est réalisé automatiquement par ToMaWork).

6.4.2.2 Illustration des collaborations possibles entre programmeur et framework

Pour illustrer l'insertion de points de reprise intermédiaires par le programmeur que nous avons décrite à la section précédente, nous considérons le code de calcul présent dans la méthode `compute` d'une tâche fictive `MockTask`. Ce code de calcul utilise des méthodes de calcul relatives à un objet (ou classe) auxiliaire `Aux`, et il est présent à la figure 6.13.

Supposons que la méthode `method1` (cf. Figure 6.13, l. 3) comprenne beaucoup de calculs et que par conséquent elle dure très longtemps ; dans ces conditions il est avantageux de réaliser un point de reprise de la valeur de la variable `i` (cf. Figure 6.14a, l. 7). Si le programmeur désire réaliser un point de reprise après l'appel de la méthode `method2`, il doit créer un objet « `serializable` » contenant les valeurs de `i`, `j` (l'itérateur de boucle) et `sum` (l'accumulateur). La figure 6.14b comprend la définition d'une la classe nommée `MySave` qui permet de créer un tel objet. L'utilisation de cette classe est illustrée à la figure 6.14a (l. 15 à 16). Enfin, supposons aussi qu'il est avantageux de sauver le résultat issu du `block2` et contenu dans la variable `b`. À ce stade, les valeurs sauvées dans l'objet `MySave` ne sont plus utiles. Il suffit donc de ne sauver que la valeur `b` (cf. Figure 6.14a, l. 20 à 21).

Comme ToMaWork n'impose pas de structure sur le code de calcul, il revient au programmeur de spécifier la manière de reprendre l'exécution à partir d'un point de reprise. Le code correspondant doit être intégré à la méthode :

– `R compute(WorkerCheckpoint c)`

```
1  public MockResult compute(MockTask t)
2  {
3    int i = Aux.method1(t);
4
5    int sum = 0;
6
7    for (int j=0; j<5; j++)
8    {
9      sum += Aux.method2(i);
10   }
11
12   boolean b = Aux.block2(sum);
13
14   MockResult br = Aux.operation3(b);
15
16   return br;
17 }
```

FIGURE 6.13 – Code source de la méthode de calcul d'une tâche fictive *MockTask*.

```
1  public MockResult compute(MockTask t)
2  {
3    java.io.Serializable s;
4
5    int i = Aux.method1(t);
6    s = i;
7    checkpoint(t, 1, s);
8    // t: tâche, 1: id du point de reprise,
9    // s: résultat intermédiaire
10
11   int sum = 0;
12   for (int j=0; j<5; j++)
13   {
14     sum += Aux.method2(i);
15     s = new MySave(sum,i,j);
16     checkpoint(t, j+2, s);
17   }
18
19   boolean b = Aux.block2(sum);
20   s = b;
21   checkpoint(t, 7, s);
22
23   MockResult br = Aux.operation3(b);
24   return br;
25 }
```

(a) Code source de la méthode *compute*.

```
1  class MySave implements Serializable
2  {
3    int sum, i, j;
4    MySave(int sum, int i, int j)
5    {
6      this.sum = sum;
7      this.i = i;
8      this.j = j;
9    }
10 }
```

(b) Code source de la classe *MySave*.

FIGURE 6.14 – Code source de la méthode de calcul *compute* de la classe *MockWorker* augmenté avec la définition de points de reprise.

Il s'agit d'une méthode abstraite à implanter par toute classe qui hérite de la classe `Worker` du paquetage `tomawork`. Le corps de cette méthode est vide lorsqu'aucun point de reprise intermédiaire n'est défini. En revanche, dans le cas décrit précédemment, le programmeur doit écrire le code de reprise associé aux points de reprises définis dans la méthode (cf. Figure 6.15, p. 111).

```
1  public MockResult compute (tomawork.WorkerCheckpoint c)
2  {
3    MockTask t = (MockTask)c.getTask();
4    java.io.Serializable s;
5
6    boolean b;
7    if (c.getCkpId() < 7)
8    {
9      int i;
10     int sum = 0;
11     int j =0;
12
13     if (c.getCkpId() == 1)
14     {
15       i = (Integer)c.getIntermediateResult();
16     }
17     else
18     {
19       MySave ms = (MySave)c.getIntermediateResult();
20       sum = ms.sum;
21       i = ms.i;
22       j = ms.j;
23     }
24
25     for (; j<5; j++)
26     {
27       sum += Aux.method2(i);
28       s = new MySave(sum,i,j);
29       checkpoint(t, j+2, s);
30     }
31
32     b = Aux.block2(sum);
33     s = b;
34     checkpoint(t, 7, s);
35   }
36   else
37   {
38     // chPt id = 7
39     b = (Boolean)c.getIntermediateResult();
40   }
41
42   MockResult br = Aux.operation3(b);
43   return br;
44 }
```

FIGURE 6.15 – Code source de la méthode de calcul d'une tâche fictive *MockTask* pour la reprise.

6.5 Résumé du framework ToMaWork

Dans ce chapitre, nous avons présenté *ToMaWork*, un framework qui résulte de l'application de MoLOToF au paradigme de parallélisation Maître-Travailleur. Ceci a abouti à deux squelettes tolérants aux pannes : l'un pour l'entité maître et l'autre pour l'entité travailleur, adaptés aussi bien aux Maître-Travailleur statiques que dynamiques. La mise en œuvre au-dessus d'un espace de tuples a permis d'obtenir des squelettes simples et bénéficiant de fonctionnalités de niveau intergiciel, par exemple celles issues de l'utilisation d'un Javaspace. Cette solution a résulté en une tolérance aux pannes standard qui maintient et sauve l'état de l'application de manière distribuée et assure la survie de l'application en cas de pannes non permanentes. Enfin, conformément au modèle MoLOToF, *ToMaWork* met en avant des collaborations entre le programmeur et le framework. La première permet au programmeur de régler, sur chacune des entités impliquées (maître, travailleurs, espace de tuples), le niveau de tolérance aux pannes desiré ; la seconde permet au programmeur de définir des points de reprise intermédiaires dans la méthode de calcul des travailleurs.

Chapitre 7

Évaluation de nos frameworks

Aux chapitres 5 et 6 nous avons respectivement présenté les frameworks FT-GReLoSSS et ToMaWork. Ces derniers résultent de l'application de MoLOToF, notre modèle pour le développement d'applications de calcul distribuées tolérantes aux pannes. L'application de MoLOToF a conduit à des squelettes tolérants aux pannes qui sont spécifiques aux paradigmes de programmation parallèles utilisés par les frameworks : SPMD pour FT-GReLoSSS et Maître-Travailleur (MT) pour ToMaWork. Cette approche résulte de la volonté de faciliter l'écriture d'applications dans les paradigmes de programmation parallèle considérés (SPMD et MT) et de permettre l'ajout d'une tolérance aux pannes efficace (et portable).

Dans ce chapitre, nous nous intéressons (1) à la vérification du fonctionnement correct des frameworks FT-GReLoSSS et ToMaWork ; (2) à l'évaluation de la facilité de développement ainsi qu'à (3) l'évaluation de l'efficacité de la tolérance aux pannes. À la section 7.1, nous explicitons les trois points énoncés précédemment et nous présentons plus en détail la méthodologie d'évaluation ainsi que la manière de réaliser nos mesures. À la section 7.2 nous présentons les résultats d'évaluation de la facilité de développement pour nos deux frameworks. Les sections 7.3, 7.4 et 7.5 sont consacrées à l'étude des performances de la tolérance aux pannes de FT-GReLoSSS.

7.1 Environnement et méthode d'évaluation

Dans le cadre de l'évaluation de nos frameworks, nous avons réécrit des applications existantes en utilisant nos frameworks. Pour FT-GReLoSSS, nous avons réécrit les applications *Matmult* et *Jacobi*, que nous avons décrites au chapitre 5. Il en a été fait de même avec les applications Pi et NQueens pour ToMaWork (cf. Chapitre 6). Ainsi, il existe deux versions pour chaque application considérée : une version sans framework que nous désignerons par *application sans framework* et une version avec framework que nous désignerons par *application avec framework*. L'application sans framework et l'application avec framework utilisent la même technologie. Ainsi, l'application de Jacobi sans framework et celle avec framework sont toutes les deux écrites avec C++ et MPI. Il en est de même pour toutes les applications utilisées pour évaluer ToMaWork : elles sont écrites en Java et utilisent la technologie JavaSpaces.

L'ensemble des expériences que nous avons réalisées consistent en une comparaison entre l'application avec framework et l'application sans framework correspondante. Pour les expériences relatives à la performance de la tolérance aux pannes, l'application sans framework a été munie d'une solution de tolérance aux pannes de niveau système provenant de recherches d'autres équipes. Cette dernière est alors comparée à la solution de tolérance aux pannes de niveau applicatif de l'application avec framework.

7.1.1 Vérification du fonctionnement correct des frameworks

Pour vérifier le fonctionnement correct des frameworks nous avons exécuté les applications implantées avec nos frameworks en l'absence et en présence de pannes. Dans le scénario en l'absence de pannes, l'application est simplement exécutée avec les mécanismes de tolérance aux pannes activés. Dans le scénario en présence de pannes, l'exécution de l'application est interrompue suite à des pannes que nous provoquons. L'interruption est suivie d'une reprise sur panne manuelle.

Dans le cas de FT-GReLoSSS, le mécanisme de tolérance aux pannes est tel que toute l'application est interrompue suite à n'importe quelle panne de processus. Ceci correspond au comportement par défaut des bibliothèques MPI que nous avons conservé dans FT-GReLoSSS. Par ailleurs, pour évaluer FT-GReLoSSS dans des cas variés, nous l'avons testé avec des pannes de processus et des pannes temporaires de machines.

Dans le cas de ToMaWork, ce dernier met en jeu en plus du processus Maître et des processus Travailleurs, les services de Javaspace, d'annuaire, et de gestionnaire de transactions de Jini (cf. Section 7.1.5). Nous avons provoqué des pannes temporaires sur chacune de ces entités de manière individuelle. Grâce à l'indépendance entre les tâches de calcul et l'architecture sous-jacente qui implique un couplage faible entre les processus de l'application, l'occurrence de pannes n'aboutit pas à l'arrêt de toute l'application comme pour FT-GReLoSSS. Ainsi, suite à une (ou plusieurs) panne(s) l'application réussit à avancer dans son exécution jusqu'à un certain point. Par exemple, la panne du service de transaction empêche le Maître et les Travailleurs de réaliser des opérations sur le Javaspace. Ceci perdure jusqu'à ce que le service de transaction soit rétabli. Le même comportement s'observe pour une panne du Javaspace. La panne de l'annuaire empêche les Travailleurs (ou le Maître) qui effectuent une reprise sur panne de rejoindre l'application jusqu'à ce que l'annuaire soit rétabli. En effet, sans annuaire, la découverte du Javaspace est compromise.

Il est intéressant de noter que les services de Javaspace, d'annuaire et de gestionnaire de transation peuvent être rendus *Activatable* selon la terminologie Java RMI. Cela signifie qu'ils peuvent être automatiquement redémarrés en cas de panne temporaire de la JVM sur laquelle ils s'exécutent. Pour y parvenir il suffit simplement de changer un paramètre dans les fichiers de configuration des services ciblés. Cette fonctionnalité n'a pas été évaluée.

Dans tous les cas experimentés, nous avons comparé le résultat fourni par l'application avec framework à celui de l'application sans framework. Ces expériences ont permis de vérifier le bon fonctionnement de nos frameworks. Le fonctionnement correct a été confirmé dans les nombreuses autres expériences menées pour évaluer l'efficacité de nos frameworks en l'absence et en présence de tolérance aux pannes. Ces expériences sont détaillées davantage dans les sections suivantes.

7.1.2 Évaluation de la facilité de développement

Pour évaluer la facilité de développement avec FT-GReLoSSS et ToMaWork, nous procédons au décompte du nombre de lignes de code source (ou *Source Lines Of Code (SLOC)*) nécessaire pour coder nos applications *benchmark* en utilisant nos *frameworks*. Ce nombre de lignes, qui comprend uniquement les lignes ajoutées par le programmeur, est ensuite comparé au nombre de lignes dans l'application sans framework.

Le décompte du nombre de lignes peut se faire de manière physique ou logique [102, 161]. Les lignes qui découlent de ce décompte sont appelées respectivement des lignes (de code source) physiques et des lignes logiques. Un décompte physique revient à compter le nombre total de lignes de code source dans le programme. D'habitude ceci comprend en plus des lignes de codes, les lignes de commentaires et les lignes vides (ou blanches). Le nombre de lignes physiques est

souvent utilisé pour obtenir l'ordre de grandeur d'une application. En revanche, un décompte logique est plus subtil et cherche plutôt à compter le nombre d'instructions. En particulier, ce décompte ne comprend ni les lignes de commentaires ni les lignes vides et surtout, il est moins dépendant du formatage du code et du style de programmation. Dans notre évaluation nous fournissons les résultats des nombres de lignes obtenus selon les deux métriques que nous utilisons pour comparer les applications sans framework aux applications avec framework. Il est important de garder à l'esprit que ces métriques demeurent des indicateurs qui possèdent leur défauts. Par exemple, elles sont incapables de distinguer les lignes de code source qui présentent une difficulté algorithmique de celles qui n'en présentent pas.

De nombreux outils existent pour réaliser ce type de mesures sur les lignes de code. Cependant, ils ne sont pas équivalents notamment pour le décompte des lignes de code source logiques et ce, à cause de l'absence d'un standard de décompte universel [102]. Dans notre évaluation nous utilisons l'outil `Unified CodeCount` ((`UCC`) [152] qui est développé à l'Université de Californie du Sud). Il permet de compter aussi bien des lignes physiques que des lignes logiques et supporte différents langages de programmation dont le C/C++ et le Java. Les nombres de lignes physiques et logiques fournis par `UCC` considèrent les directives de compilation, les déclarations de variables et les autres instructions d'exécution. Les commentaires et les lignes vides ne sont pas comptabilisés.

7.1.3 Description des expériences d'évaluation des performances des frameworks

L'évaluation des performances de nos frameworks comprend plusieurs expériences. Nous évaluons d'abord la *performance des frameworks en l'absence de tolérance aux pannes* (cf. Expérience 1 ci-après) puis nous nous intéressons à l'évaluation des *performances en présence de tolérance aux pannes* : nous distinguons les *performances en l'absence de pannes* (cf. Expérience 2 ci-après) des *performances en présence de pannes* (cf. Expérience 3 ci-après).

Expérience 1 : évaluation en l'absence de tolérance aux pannes

Pour l'évaluation en l'absence de tolérance aux pannes, les mécanismes de tolérance aux pannes de l'application avec framework sont désactivés. Avec FT-GReLoSSS, cela revient à n'effectuer aucun point de reprise pendant toute la durée de l'exécution de l'application. Avec ToMaWork, aucune entité impliquée ne réalise de point de reprise, et le service de transaction est désactivé. Nous procédons alors à la comparaison des temps d'exécution selon deux configurations :

- **Configuration 1 :** les applications avec framework sont exécutées sans tolérance aux pannes et les temps réalisés sont comparés à ceux des applications sans framework ;
- **Configuration 2 :** les temps d'exécution obtenus par les applications avec framework dans la configuration précédente sont à présent comparés à ceux de leurs équivalents sans framework. Ces derniers sont associés à un SPRD de niveau système qui provient de travaux extérieurs à notre équipe. Au cours des exécutions des applications sans framework, les mécanismes de tolérance aux pannes du SPRD associé sont désactivés (*i.e.* : ceci revient le plus souvent à ne réaliser aucun point de reprise).

La première configuration permet de situer l'application avec framework par rapport à la même application sans framework tandis que la seconde permet de déceler d'éventuels surcoûts résultant de l'utilisation de la solution de tolérance aux pannes de niveau système. Habituelle-

ment, ces dernières sont peu intrusives et introduisent un surcoût faible lorsqu'elles sont utilisées sans réaliser de points de reprise.

Cette première expérience fournit un étalonnage des solutions comparées et constitue donc une étape préliminaire à la réalisation des expériences en présence de tolérance aux pannes dans lesquelles nous considérons d'abord la situation où aucune panne ne survient (cf. Expérience 2) puis celle où des pannes surviennent (cf. Expérience 3).

Expérience 2 : évaluation en présence de tolérance aux pannes et en l'absence de pannes

Une des particularités des frameworks FT-GReLoSSS et ToMaWork est d'impliquer le programmeur de l'application dans le processus de tolérance aux pannes afin d'obtenir une tolérance aux pannes efficace (fondée sur la réalisation de points de reprise minimaux). Ceci s'accomplit principalement par le biais d'un contrôle (1) sur les données à sauver dans les points de reprise et (2) sur la fréquence de réalisation des points de reprise.

Dans cette expérience, nous cherchons donc à évaluer l'impact de l'implication du programmeur de l'application dans le processus de tolérance aux pannes. Pour ce faire, nous nous intéressons aux tailles des points de reprise résultant de l'implication du programmeur, puis à leur impact sur l'exécution :

- **Configuration 1 (impact sur la taille des points de reprise) :** nous comparons les volumes de données des points de reprise sauvés sur disque pour l'application avec framework et avec l'implication du programmeur à ceux sauvés (1) par l'application avec framework et sans implication du programmeur, et (2) par l'application sans framework et utilisant une solution de tolérance aux pannes de niveau système.
- **Configuration 2 (impact sur les temps d'exécution) :** nous comparons les temps d'exécution réalisés sous l'effet d'un nombre de points de reprise grandissant (1) par les applications avec framework (et avec implication du programmeur) à ceux réalisés (2) par les applications sans framework (avec un SPRD de niveau système).

La première configuration évalue le gain en terme de taille de points de reprise réalisés résultant d'une collaboration du programmeur avec le SPRD. Quant à la deuxième configuration, elle permet dans un premier temps de montrer l'impact d'une telle intervention sur l'application avec framework. Dans un second temps, elle permet d'évaluer l'efficacité de la réalisation des points de reprise du framework par rapport à des solutions reposant sur un SPRD de niveau système.

Expérience 3 : évaluation en présence de tolérance aux pannes et de pannes

Dans cette expérience nous cherchons à évaluer la performance de la tolérance aux pannes de nos frameworks en présence de pannes :

- **Configuration 1 (évaluation du temps de reprise) :** il s'agit du temps nécessaire pour que l'application victime de la panne retrouve le cours normal de son exécution. Les temps mesurés ne comprennent pas le temps de détection de la panne. Ceci résulte du fait que nos frameworks ne disposent pas de mécanisme(s) de détection et de reprise automatiques. Ceci vaut aussi pour les SPRD auxquels nous nous comparons. Ces derniers, supposent que la détection est réalisée par un tiers à l'application et au SPRD. Les temps de reprise réalisés par l'application avec framework sont comparés aux temps réalisés par l'application sans framework munie d'un SPRD de niveau système.

– **Configuration 2 (évaluation du temps maximal de calcul perdu) :** il s'agit du temps écoulé entre la date de réalisation du dernier point de reprise et la date d'occurrence de la panne. Les frameworks et les SPRD sont alors comparés sur leur capacité à perdre le moins de temps de calcul possible en présence de pannes.

Pour des raisons de manque de temps, l'évaluation de ToMaWork a été moins poussée que celle de FT-GReLoSSS durant cette thèse. Notamment, nous n'avons pas pu réaliser de comparaison avec des solutions existantes de tolérance aux pannes (de niveau système). Une autre comparaison intéressante et complémentaire à celle utilisée aurait consisté à se comparer avec des systèmes de tolérance aux pannes de niveau framework (ou applicatif). Cela dit, leur prise en main (installation, déploiement, portage d'application, etc.) n'est souvent pas immédiate et leur utilisation dans nos évaluations aurait requis une connaissance approfondie de leur fonctionnement. Par ailleurs, ces systèmes ne sont pas toujours disponibles, et, lorsqu'ils le sont, il n'est pas facile de trouver du support.

7.1.4 Méthodologie de mesure

Dans nos expériences nous mesurons des tailles de points de reprise sur disque et des temps d'exécution. La mesure de la taille des points de reprise s'effectue avec la commande `UNIX` « `du -h` ». L'option « `-h` » permet d'afficher les tailles en unités compréhensibles par l'utilisateur et non pas en nombre de blocs.

Pour la mesure des temps d'exécution, nous effectuons des mesures à l'intérieur des applications (ou mesures internes) avec la fonction `gettimeofday` en environnement C et la méthode `currentTimeMillis` (de la classe `System`) en environnement Java. Nous accompagnons ces mesures internes d'une mesure externe grâce à la commande `UNIX` « `time` ». L'ensemble des temps fournis sont acquis de manière interne sauf dans le cas de l'évaluation du temps de reprise sur panne où nous avons fait le choix d'utiliser `time` plutôt qu'une approche de mesure interne. L'utilisation d'une mesure interne aurait requis de modifier les applications de manière à afficher le temps juste avant la panne, qui aurait donc du être déclenchée depuis l'application après avoir réalisé un point de reprise. Au final, cela aurait fortement contraint les expérimentations, et aurait été très complexe dans le cas des SPRD de niveau système. En comparaison, avec des mesures externes, il suffit de mesurer (1) le temps entre le moment où l'application débute et le moment où elle est interrompue par une panne, puis (2) le temps entre le moment où l'application reprend et la fin l'application. Ce deuxième temps est mesuré ensuite plusieurs fois de manière à s'assurer de la stabilité des mesures. Cette façon de procéder a donc été préférée à celle de mesure internes en raison de sa simplicité de mise en œuvre, et de son très faible niveau d'intrusivité.

Les temps fournis constituent des moyennes réalisées à partir de 5 mesures. D'habitude les mesures réalisées sur la grappe Intercell — notre plate-forme physique d'expérimentation (cf. Section 7.1.5) — sont (très) stables comme en témoigne le tableau 7.1. Ce dernier contient la moyenne, le minimum, le maximum, l'écart type ainsi que l'écart type en pourcentage de la moyenne sur un ensemble de 10 mesures de temps d'exécution. Les exécutions ont été réalisées avec l'application *Matmult* de FT-GReLoSSS sur 64 processeurs et pour des matrices de taille 32768×32768. Au cours de l'exécution, l'application réalise 7 points de reprise de manière à ne pas négliger dans nos mesures les entrées/sorties (E/S) disques, qui peuvent être à l'origine d'écarts importants dans les mesures. Classiquement, la reproductibilité des mesures est compromise lorsque la part attribuée aux E/S — aussi bien sur le réseau que sur disque — devient trop importante par rapport à la part de calculs. Mais les écarts type calculés, qui valent $4s$ (resp.

4.63s) pour la mesure interne (resp. externe) pour des temps d'exécution moyens de plus 1100s, témoignent d'une bonne stabilité de la grappe Intercell et des instruments de mesure.

Toutefois, il nous est arrivé d'obtenir des résultats de mesures très éloignés de la moyenne. L'origine de tels comportements nous est inconnue, mais un redémarrage de l'ensemble des machines impliquées a suffi à rétablir le système. Les mesures ont alors été refaites, et nous avons décidé d'évincer les résultats singuliers obtenus avant le redémarrage des machines.

La plupart de nos mesures exhibent une stabilité similaire à celle des mesures de la table 7.1. Dans la suite, sauf indication contraire, nous présenterons des temps moyens réalisés sur 5 mesures et implicitement associés à des écarts types faibles.

Taille des matrices	Nombre de nœuds	Outil de mesure	Temps d'exéc. t (s)			$\sigma(t)(s)$	$\sigma(t)$ en % de t_{moy}
			t_{min}	t_{moy}	t_{max}		
32768×32768	64	`gettimeofday`	1112	1116	1125	3.98	0.36
		`time -p`	1118	1123	1134	4.63	0.41

Table 7.1 – Illustration de la stabilité de la précision des mesures réalisées sur la grappe Intercell : statistiques obtenues sur 10 exécutions de l'application *Matmult* dévelopée avec le framework FT-GReLoSSS, et en présence de la réalisation de 7 points de reprise.

7.1.5 Plate-forme d'expérimentation matérielle et logicielle

Les expérimentations ont été menées sur la grappe `InterCell` installée sur le campus de Supélec à Metz [69]. La mise en place d'`InterCell` s'inscrit dans le *CPER Lorrain* (2007-2013) (Contrat de Projets État-Région) et fait parti du Pôle de Recherche Scientifique et Technologique MISN (Modélisations, Informations et Systèmes Numériques) [38]. Elle a été subventionnée par l'INRIA et la Région Lorraine.

La grappe Intercell , qui a été fournie par la société *CARRI Systems*, comporte 256 nœuds de calcul reliés par un réseau `Gigabit Ethernet`. Chaque nœud héberge un processeur double cœur. Les deux cœurs se partagent `4 Go` de mémoire. Les processeurs sont des `Intel Xeon-3075` cadencés à `2.66 GHz` avec un bus de communication à `1333 MHz`. L'interconnexion est assurée par un switch `Cisco Catalyst 6509`. Le système d'exploitation des nœuds est une Fedora Core 8 en 64-bit.

La présence de NFS sur les nœuds de calcul permet le partage de données depuis les comptes des utilisateurs qui sont « montés » sur les nœuds de calcul. Cette stratégie facilite le déploiement d'applications interactives ainsi que l'utilisation de la grappe par des non-spécialistes du calcul parallèle. En revanche, NFS est complètement inadapté pour des E/S intensives que génèrent par exemple l'écriture (ou la lecture) de points de reprise. Dans ce dernier cas un système de fichier parallèle, qui pour le moment est absent d'`InterCell`, aurait été beaucoup plus approprié. Ainsi, dans toutes nos expériences, les points de reprise sont écrits (resp. lus) sur (resp. depuis) les disques durs locaux aux nœuds sur lesquels s'exécutent les processus de l'application distribuée.

Enfin, la présence de processeurs bi-cœurs sur chaque nœud de calcul permet d'envisager l'exécution de deux processus applicatifs par nœud. Certes cette stratégie permet de bénéficier de plus d'unités de calcul mais n'est pas toujours intéressante. En particulier, pour un nombre de processus P fixé, les performances obtenues en exécutant ces processus sur P nœuds et sur $P/2$

nœuds ne sont pas toujours équivalentes [85]. Pour notre part, dans les expériences présentées ci-après les applications étaient déployées à raison d'un processus applicatif par nœud de calcul.

Dans le cadre des expériences relatives au framework FT-GReLoSSS, ce dernier est comparé à DMTCP 1.06 r254 [4, 42] et LAM/MPI 7.1.4. [26, 79]. Open MPI 1.3.3 (OMPI) [51, 107] est utilisé pour réaliser les mesures de performance avec FT-GReLoSSS et DMTCP. Sauf mention contraire, FT-GReLoSSS est utilisé avec l'optimisation sur la taille des points de reprise et la bibliothèque d'E/S standard en mode binaire. Parmi les stratégies d'exécution du plan de routage disponibles, nous utilisons celle qui plannifie toutes les communications en même temps pour les recouvrir au maximum (cf. Section 5.6.2, p. 91).

Dans le cadre des expériences relatives au framework ToMaWork, nous avons utilisé le Jini Starter Kit v2.1 (JSK) avec des JVM 1.6.0 de Sun Microsystems [17]. Le JSK [18] offre plusieurs services reposant sur et étendant la technologie Java pour permettre la construction de systèmes distribués. Parmi les services proposés, nous avons utilisé le service d'annuaire (`reggie`), le service de transactions (`mahalo`) et le service de mémoire partagée virtuelle Javaspace (`outrigger`).

7.2 Évaluation de la facilité de développement

7.2.1 Résultats pour FT-GReLoSSS

	Type de lignes	Sans FT-GReLoSSS	Avec FT-GReLoSSS	Surcoût absolu	Surcoût relatif (%)
Matmult	physique	258	295	+37	+14.3
	logique	168	186	+18	+10.7
Jacobi	physique	293	316	+29	+7.8
	logique	179	190	+11	+6.1

Table 7.2 – Surcoût introduit par FT-GReLoSSS en terme de nombre de lignes physiques et logiques de code source pour deux applications de test.

Le tableau 7.2 reporte le nombre de lignes physiques et logiques décomptées par le logiciel `Unified CodeCount` [152] pour les applications *Matmult* et *Jacobi*. Le nombre de lignes physiques pour les applications avec framework est 7.8% à 14.3% plus élevé que celui des applications sans framework. Celui des lignes logiques est 6.1% à 10.7% plus élevé.

Les surcoûts observés s'expliquent par la présence dans l'application avec framework d'un certain nombre d'opérations qui sont inexistantes dans l'application sans framework et qui correspondent à : (1) l'initialisation et la finalisation de l'environnement tolérant aux pannes, (2) l'interfaçage avec la représentation de la structure de données distribuée du framework et (3) l'interfaçage avec le squelette tolérant aux pannes (cf. Section 5.3.4 (p. 72) et section 5.4.2 (p. 80)).

Ces surcoûts peuvent sembler importants à première vue mais correspondent à un nombre de lignes logiques (ou nombre d'instructions) compris entre 11 et 18. Un tel nombre n'est pas énorme d'autant plus que parmi ces instructions, plusieurs ne présentent aucune difficulté algorithmique.

17. Sun Microsystems a été rachetée par ORACLE au début de 2010.

18. Il s'agit d'un projet initié par la compagnie Sun Microsystems et dont le développement a été repris par la fondation `Apache` sous le nom d'`Apache River`.

En fait, seules les instructions des fonctions de partitionnement peuvent requérir un effort plus important de la part du programmeur. Enfin, dans le cas d'une grosse application, la part du code dédiée aux calculs devient significativement plus importante (en nombre de lignes de code) que la part de code que doit ajouter le programmeur pour bénéficier de la tolérance aux pannes de FT-GReLoSSS.

FT-GReLoSSS est donc à l'heure actuelle un peu plus verbeux par rapport à des solutions de tolérance aux pannes (semi-)transparentes. Mais cette verbosité est principalement constituée de nombreuses instructions qui ne présentent aucune difficulté algorithmique et qui, par ailleurs, demeurent relativement constantes d'une application à l'autre. Alors que ceci peut sembler inacceptable pour certains programmeurs, les bénéfices de cette approche apparaissent à l'exécution, notamment lors de la comparaison avec des solutions transparentes (cf. Section 7.4.2, p. 128).

7.2.2 Résultats pour ToMaWork

Le tableau 7.3 reporte le nombre de lignes de code logiques décomptées par l'outil `Unified CodeCount` pour l'application NQueens qui calcule le nombre de solutions au problème classique des N reines. Ici, le nombre de lignes physiques et logiques sont respectivement 19.2% et 24.5% plus faibles pour l'application avec framework par rapport à celle sans framework.

La prise en charge automatique des opérations de découverte du Javaspace permet d'expliquer une partie de ces surcoûts négatifs. Il s'agit en effet, d'une série d'instructions systématiques pour un programmeur qui utilise la technologie JavaSpaces. L'autre partie des surcoûts négatifs tient au fait que ToMaWork intègre entièrement le modèle de parallélisation au sein de ses squelettes (cf. Section 6.2.1, p. 95). Ce modèle décrit la manière d'intéragir du Maître et des Travailleurs avec le Javaspace.

Par rapport à FT-GReLoSSS, ToMaWork réussit à mieux découpler la structure de son paradigme du code du programmeur. Cette meilleure séparation est rendue possible d'une part grâce à la simplification de la nature des données échangées (tâches et résultats), qui, en outre, bénéficient d'un mécanisme de sérialisation transparent car inhérent au langage Java. D'autre part, cette séparation est facilitée par la présence d'un schéma de communications très peu variable entre les différentes entités impliquées dans le paradigme Maître-Travailleur. Dans un langage comme C++ — qui est utilisé par FT-GReLoSSS — il n'existe pas de mécanisme de sérialisation transparent des données ce qui nous mène à utiliser une interface pour supporter la diversité des structures de données utilisées. Ensuite, le schéma de communications dans le paradigme de parallélisation SPMD n'est pas aussi uniforme que dans celui du Maître-Travailleur. FT-GReLoSSS permet de paralléliser aussi bien des applications mettant en jeu des communications par échange de frontières que par circulation de données. La flexibilité offerte par FT-GReLoSSS rend plus difficile le type de factorisation atteint par ToMaWork avec le paradigme Maître-Travailleur.

De manière similaire à FT-GReLoSSS, les surcoûts négatifs en faveur de ToMaWork auront tendance à devenir proportionnellement moins importants pour des applications plus grosses. Néanmoins, l'interface proposée par ToMaWork présente deux grands avantages. D'une part elle demeure plus simple à utiliser par un public non spécialiste en développement d'applications de calcul distribué. D'autre part, la facilité et la flexibilité offertes en matière de tolérance aux pannes devraient intéresser même les spécialistes.

	Type de lignes	Sans ToMaWork	Avec ToMaWork	Surcoût absolu	Surcoût relatif (%)
NQueens	physique	625	505	−120	−19.2
	logique	408	308	−100	−24.5

TABLE 7.3 – Surcoût introduit par *ToMaWork* en terme de nombre de lignes physiques et logiques de code source pour l'application NQueens.

7.3 Performances de FT-GReLoSSS en l'absence de tolérance aux pannes

Dans cette section nous évaluons les surcoûts induits par l'utilisation du framework FT-GReLoSSS. Les surcoûts concernent les performances du framework et on s'intéresse à l'évaluation du surcoût de FT-GReLoSSS à l'exécution lorsque la tolérance aux pannes est désactivée (*i.e.* : lorsqu'aucun point de reprise n'est réalisé) (cf. Expérience 1, Configuration 1 à la section 7.1.3). Dans ce but, nous comparons en premier lieu les temps d'exécution d'applications utilisant FT-GReLoSSS à ceux réalisés par l'application équivalente mais sans FT-GReLoSSS ; dans les deux cas l'implantation MPI est la même. En second lieu, nous nous comparons aux temps réalisés en utilisant deux autres méthodes transparentes de tolérance aux pannes (cf. Expérience 1, Configuration 2 à la section 7.1.3). La première est LAM/MPI (7.1.4) une implantation de MPI intégrant un mécanisme de réalisation de points de reprise. La seconde est DMTCP un système indépendant de la bibliothèque MPI utilisée qui permet également de réaliser des points de reprise distribués. DMTCP est utilisé avec la même implantation MPI que celle utilisée par *FT-GReLoSSS* (*i.e.* : OMPI 1.3.3). Ces expériences permettent d'obtenir les temps de réference des différentes solutions de tolérance aux pannes utilisées.

Taille des matrices	Nombre de nœuds	T_{exec} (secondes)		Surcoût relatif (%)
		OMPI	FT-GReLoSSS	
16384×16384	4	2027	2027	0.0
	8	1025	1027	0.3
	16	522	526	0.7
	32	274	277	0.9
32768×32768	32	2107	2113	0.3
	64	1094	1103	0.8
	128	597	609	1.9
	256	352	362	3.0
65536×65536	64	8405	8439	0.4
	128	4444	4469	0.6
	256	2406	2445	1.6

TABLE 7.4 – Surcoût introduit par FT-GReLoSSS sans tolérance aux pannes pour l'application *Matmult* avec OMPI.

À cause de la structuration qu'il impose et de sa généricité, FT-GReLoSSS introduit un

Taille des matrices	Nombre de nœuds	T_{exec} (secondes)		Surcoût relatif (%)
		OMPI	DMTCP	
16384×16384	4	2027	2023	−0.2
	8	1025	1017	−0.8
	16	522	523	0.2
	32	274	272	−0.7
32768×32768	32	2107	2106	0.0
	64	1094	1094	0.0
	128	597	597	0.0
	256	352	352	0.0
65536×65536	64	8405	8547	1.3
	128	4444	4558	2.2
	256	2406	2408	0.1

TABLE 7.5 – Surcoût introduit par DMTCP avec OMPI et sans tolérance aux pannes pour l'application *Matmult* par rapport à OMPI.

Taille des matrices	Nombre de nœuds	T_{exec} (secondes)		Surcoût relatif (%)
		OMPI	LAM/MPI	
16384×16384	4	2027	2028	0.1
	8	1025	1029	0.4
	16	522	528	1.1
	32	274	282	2.8
32768×32768	32	2107	2137	1.4
	64	1094	1120	2.4
	128	597	631	5.6
	256	352	385	9.4
65536×65536	64	8405	8553	1.8
	128	4444	4633	4.2
	256	2406	2561	6.5

TABLE 7.6 – Surcoût introduit par LAM/MPI sans tolérance aux pannes pour l'application *Matmult* par rapport à OMPI.

surcoût à l'exécution. La connaissance de ce surcoût est importante pour un utilisateur potentiel du framework car elle lui donnera une idée des ralentissements auxquels il peut s'attendre. Le tableau 7.4 compare les temps d'exécution de l'application *Matmult* avec et sans FT-GReLoSSS pour des multiplications de matrices de trois tailles différentes, sur différents nombres de nœuds, et à raison d'un processus applicatif par nœud. Les petit, moyen et gros problèmes correspondent respectivement à une multiplication de matrices de tailles 16384×16384, 32768×32768 et 65536×65536. Dans ces exécutions aucun point de reprise n'est réalisé. Comme en attestent les chiffres du tableau 7.4 les surcoûts de temps d'exécution avec FT-GReLoSSS varient avec la taille du problème et la taille de la machine parallèle utilisée. Ils restent néanmoins très faibles et n'excèdent pas les 3%, valeur qui est atteinte uniquement sur les exécutions sur 256 nœuds.

Résultat. *L'utilisation de FT-GReLoSSS avec OMPI et sans réaliser de points de reprise introduit un léger surcoût ($< 3\%$) par rapport à OMPI, dans notre benchmark de produit de matrice de 1 à 256 nœuds.*

Les tableaux 7.5 et 7.6 reportent respectivement les surcoûts de DMTCP et de LAM/MPI par rapport à OMPI. Tout comme pour FT-GReLoSSS, aucun point de reprise n'est réalisé. Nous pouvons observer que DMTCP n'introduit pas de surcoût. Quant à LAM/MPI, il introduit un surcoût beaucoup plus important que FT-GReLoSSS et DMTCP. Le plus grand surcoût vaut 9.4% et est atteint sur 256 nœuds pour des tailles de matrices de 32768×32768.

Ces écarts importants observés entre LAM/MPI et OMPI nous amènent à maintenir deux références : une pour chacune des implantations MPI considérées. En particulier, dans la suite, tout calcul de surcoût relatif de temps d'exécution de LAM/MPI avec réalisation de points de reprise est calculé par rapport aux temps d'exécution de réference LAM/MPI qui ont été présentés dans la présente section. De la même manière, les surcoûts relatifs des temps d'exécution de DMTCP et FT-GReLoSSS avec réalisation de points de reprise, sont calculés par rapport aux temps d'exécution de réference de OMPI.

Ces comportements que l'on observe sur FT-GReLoSSS et LAM/MPI sont attribués à une faiblesse dans les communications. Avec FT-GReLoSSS elle provient du plan de routage des communications utilisé et plus particulièrement des recopies réalisées pour constituer un unique tampon d'émission et minimiser le nombre d'envois à destination du même processus (cf. Section 5.6.2, p. 91) quel que soit le problème traité. Une évaluation quantitative dans laquelle nous mesurons le temps passé en communications et en calculs pendant l'exécution (pour FT-GReLoSSS et pour LAM/MPI) confirme l'hypothèse des communications. Les temps passé en communications figurent au tableau 7.7. Les communications de FT-GReLoSSS sont en moyenne 12% plus importantes que pour OMPI contre 29% pour LAM/MPI comparé à OMPI. En conséquence, FT-GReLoSSS et LAM/MPI pénalisent les applications dans le cas où les communications deviennent trop importantes par rapport aux calculs.

Nous tenons également à remarquer que LAM/MPI possède une philosopie de déploiement des applications MPI qui est différente de celle d'OMPI. Le déploiement d'une application utilisant LAM/MPI nécessite le déploiement préalable d'un processus démon sur chaque nœud qui sera utilisé. Le temps mis pour ce déploiement est important mais n'a pas été pris en compte dans nos mesures. On suppose que si on n'utilisait que LAM/MPI, ces démons seraient déployés une unique fois : lors du démarrage des nœuds par exemple. Dans OMPI, le déploiement est plus

simple pour l'utilisateur (pas de déploiement préalable). Les temps que nous avons mesurés pendant nos expérimentations révèlent pour OMPI des temps de déploiements très rapides et négligeables par rapport aux durées d'exécutions de nos applications.

Taille du problème	Nombre de nœuds	T_{comms} (seconds)			Surcoût relatif (%)	
		OMPI	FT-GReLoSSS	LAM/MPI	FT-GReLoSSS	LAM/MPI
16384 × 16384	4	22.64	25.27	29.16	11.6	28.8
	8	23.33	25.37	30.15	8.7	29.2
	16	21.56	24.86	27.96	15.3	29.6
	32	23.54	26.73	30.46	13.5	29.4
32768 × 32768	32	87.37	100.38	113.14	14.8	29.5
	64	95.90	109.00	114.95	13.6	19.8
	128	97.73	101.93	124.36	4.3	27.2
	256	98.87	109.06	124.54	10.3	25.9
65536 × 65536	64	370.18	412.32	477.22	11.3	28.9
	128	364.36	413.23	464.23	13.4	27.4
	256	367.20	410.62	478.55	11.8	30.3

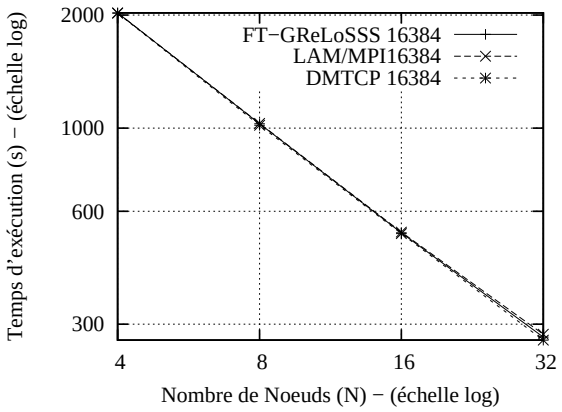
TABLE 7.7 – Comparaison des temps de communications de FT-GReLoSSS et LAM/MPI par rapport à OMPI pour l'application *Matmult*.

Résultat. *Les performances de FT-GReLoSSS sont meilleures lorsque le rapport entre le temps des communications et le temps de calculs est faible (communications proportionnellement peu importantes). Elles sont alors très proches des performances d'OMPI.*

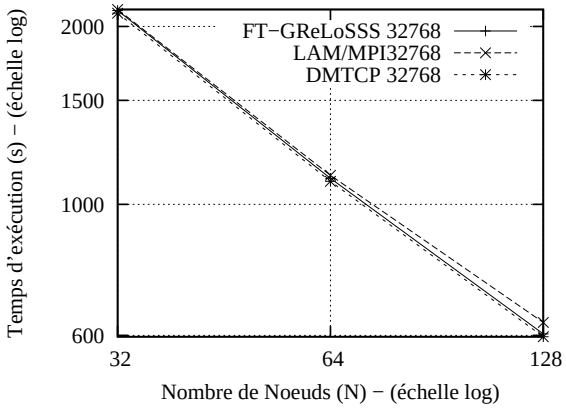
Les courbes des temps d'exécution en fonction du nombre de nœuds utilisés pour les trois différentes tailles de matrices sont présentées à la figure 7.1. Indépendamment des surcoûts introduits et qui sont visibles sur les tracés de cette figure, on remarque que les solutions considérées assurent toutes le passage à l'échelle de l'application *Matmult*. Il s'agit bien d'un benchmark représentant des applications parallèles utilisables sur un grand nombre de processeurs.

7.4 Performances de FT-GReLoSSS avec tolérance aux pannes en l'absence de pannes

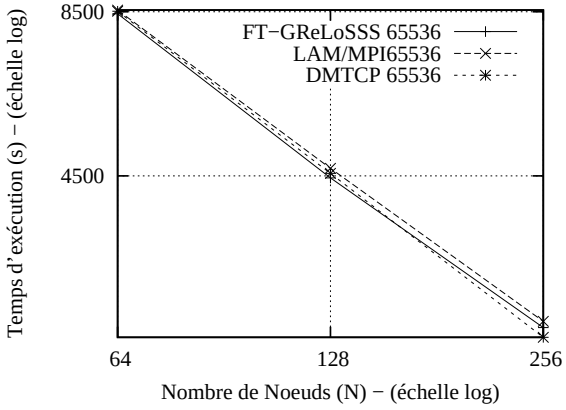
Dans cette section nous évaluons le surcoût de la tolérance aux pannes introduite par FT-GReLoSSS. Pour ce faire, nous commençons par comparer les tailles de points de reprise réalisés dans différentes situations par FT-GReLoSSS, LAM/MPI et DMTCP. Ensuite, nous comparons les temps d'exécution de FT-GReLoSSS à ceux de LAM/MPI et DMTCP pour un même nombre de points de reprise réalisés en l'absence de pannes.



(a) Taille des matrices : 16384 × 16384



(b) Taille des matrices : 32768 × 32768



(c) Taille des matrices : 65536 × 65536

FIGURE 7.1 – Comparaison des temps d'exécutions de FT-GReLoSSS, LAM/MPI et DMTCP pour différentes tailles de matrices en fonction du nombre de nœuds N.

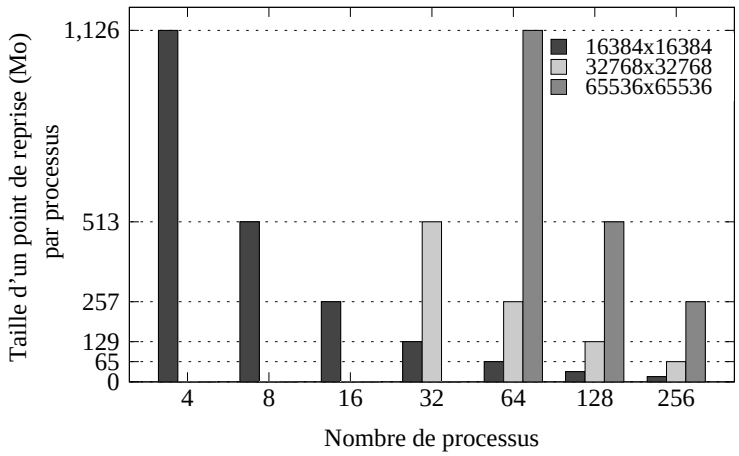


FIGURE 7.2 – Taille d'un point de reprise (en Mégaoctets) par processus, obtenue pour l'application *Matmult* avec FT-GReLoSSS et avec les optimisations sur la taille apportées par le programmeur. La taille du point de reprise est obtenue au cours d'exécutions avec différentes tailles de matrices (16384×16384, 32768×32768 ou 65536×65536) et différents nombres de processus (4 à 256).

7.4.1 Étude comparative de la taille des points de reprise

La taille des points de reprise (ou sauvegardes) est un élément important dans l'efficacité d'une solution fondée sur la réalisation de points de reprise. FT-GReLoSSS permet au programmeur de réduire la taille des points de reprise réalisés par chaque processus en utilisant sa connaissance de l'application.

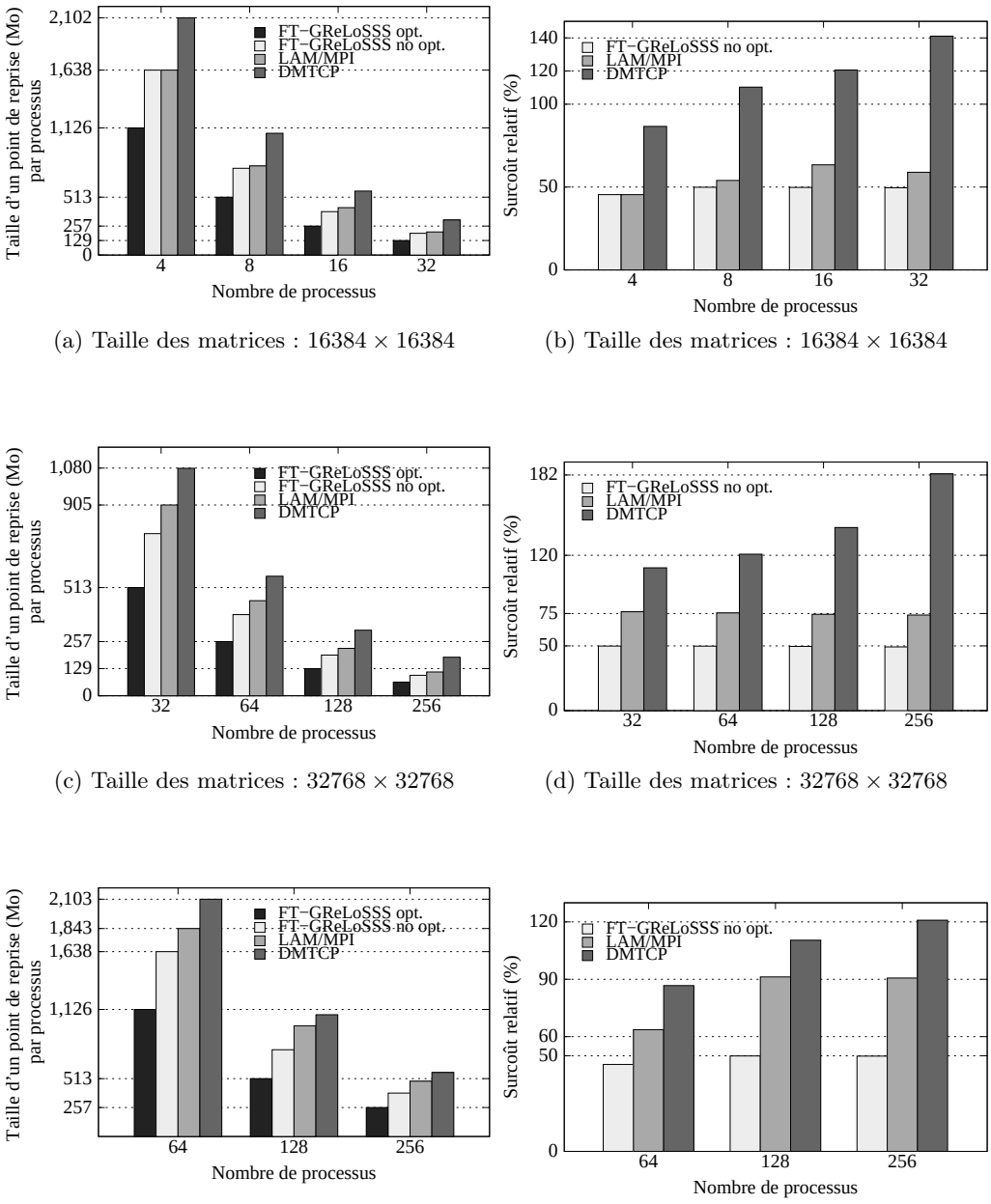
La figure 7.2 reporte les tailles des points de reprise obtenues par FT-GReLoSSS suite aux optimisations sur la taille des points de reprise décrites à la section 5.4.2 (p. 80). Ces tailles sont issues de différentes exécutions de l'application *Matmult*. Les exécutions diffèrent par la taille des matrices utilisées ainsi que par le nombre de processus utilisés. La taille reportée est celle d'un point de reprise d'un seul processus de l'application car les tailles sont identiques sur les autres processus. Les mesures ont été réalisées une seule fois car la taille des points de reprise est invariable d'une exécution à l'autre.

Pour une taille de matrice fixée, nous pouvons observer des tailles de points de reprise qui sont bien divisées par 2 lorsque le nombre de processus est doublé. Ceci est normal car les données sont équitablement réparties sur l'ensemble des processus d'après les fonctions de partitionnement.

Les diagrammes en bâtons de la figure 7.3 comparent les tailles de points de reprise obtenues par FT-GReLoSSS avec optimisations du programmeur, à celles obtenues par FT-GReLoSSS sans optimisations, par LAM/MPI et par DMTCP. La comparaison est réalisée pour les trois tailles de matrices considerées et pour différents nombres de processus.

À la figure 7.3, les diagrammes 7.3a, 7.3c, et 7.3e (sur la colonne de gauche) comparent les tailles des points de reprise réalisés par les différentes solutions envisagées pour des tailles de matrices de 16384×16384, 32768×32768 et 65536×65536 respectivement. Les diagrammes 7.3b, 7.3d, et 7.3f (sur la colonne de droite) affichent le surcoût relatif (en %) de FT-GReLoSSS sans optimisations, de LAM/MPI et de DMTCP par rapport à FT-GReLoSSS avec optimisations.

Les tailles des points de reprise de FT-GReLoSSS sans optimisations sont 50% plus importantes que celles de FT-GReLoSSS avec optimisations. La constance de cette différence découle



(a) Taille des matrices : 16384 × 16384

(b) Taille des matrices : 16384 × 16384

(c) Taille des matrices : 32768 × 32768

(d) Taille des matrices : 32768 × 32768

(e) Taille des matrices : 65536 × 65536

(f) Taille des matrices : 65536 × 65536

FIGURE 7.3 – Comparatif entre FT-GReLoSSS avec optimisations (opt.), FT-GReLoSSS sans optimisation (no opt.), LAM/MPI et DMTCP sur les tailles de points de reprise réalisés pour l'application *Matmult*. À gauche, représentation de la taille d'un point de reprise (en Mégaoctets) réalisé par processus, pour des exécutions sur différents nombres de nœuds et avec différentes tailles de matrices. À droite, représentation du surcoût relatif (en %) des tailles de points de reprise de FT-GReLoSSS sans optimisation, LAM/MPI et DMTCP par rapport à FT-GReLoSSS avec optimisation.

du fait que les points de reprise réalisés dans les deux cas ont une taille constante. Ce n'est pas le cas de LAM/MPI et de DMTCP pour lesquels les tailles mesurées sont plus variables mais demeurent (au moins) 50% plus importantes que celles de FT-GReLoSSS avec optimisations. Cette différence dans les tailles obtenues provient d'une part de l'absence des optimisations présentes dans FT-GReLoSSS ; et d'autre part de l'approche de niveau système (de LAM/MPI et DMTCP) laquelle sauvegarde tout le processus. Notamment, une sauvegarde au moment des communications aura tendance à donner un point de reprise plus volumineux étant donné qu'il contiendra aussi des données présentes dans les tampons de communications réseau. Pour une raison qui nous échappe, les tailles des points de reprise obtenues avec DMTCP sont beaucoup plus importantes que celles de LAM/MPI et peuvent atteindre le double de celles réalisées par FT-GReLoSSS avec optimisations.

Ainsi, les optimisations permettent à FT-GReLoSSS dans le cas de *Matmult* d'obtenir les points de reprise les plus petits en taille. Ceci remplit l'objectif de nos développements sur la taille des points de reprise.

Résultat. *Notre benchmark de* Matmult *montre que La contribution du programmeur dans les applications développées avec FT-GReLoSSS, peut réduire considérablement la taille des points de reprise.*

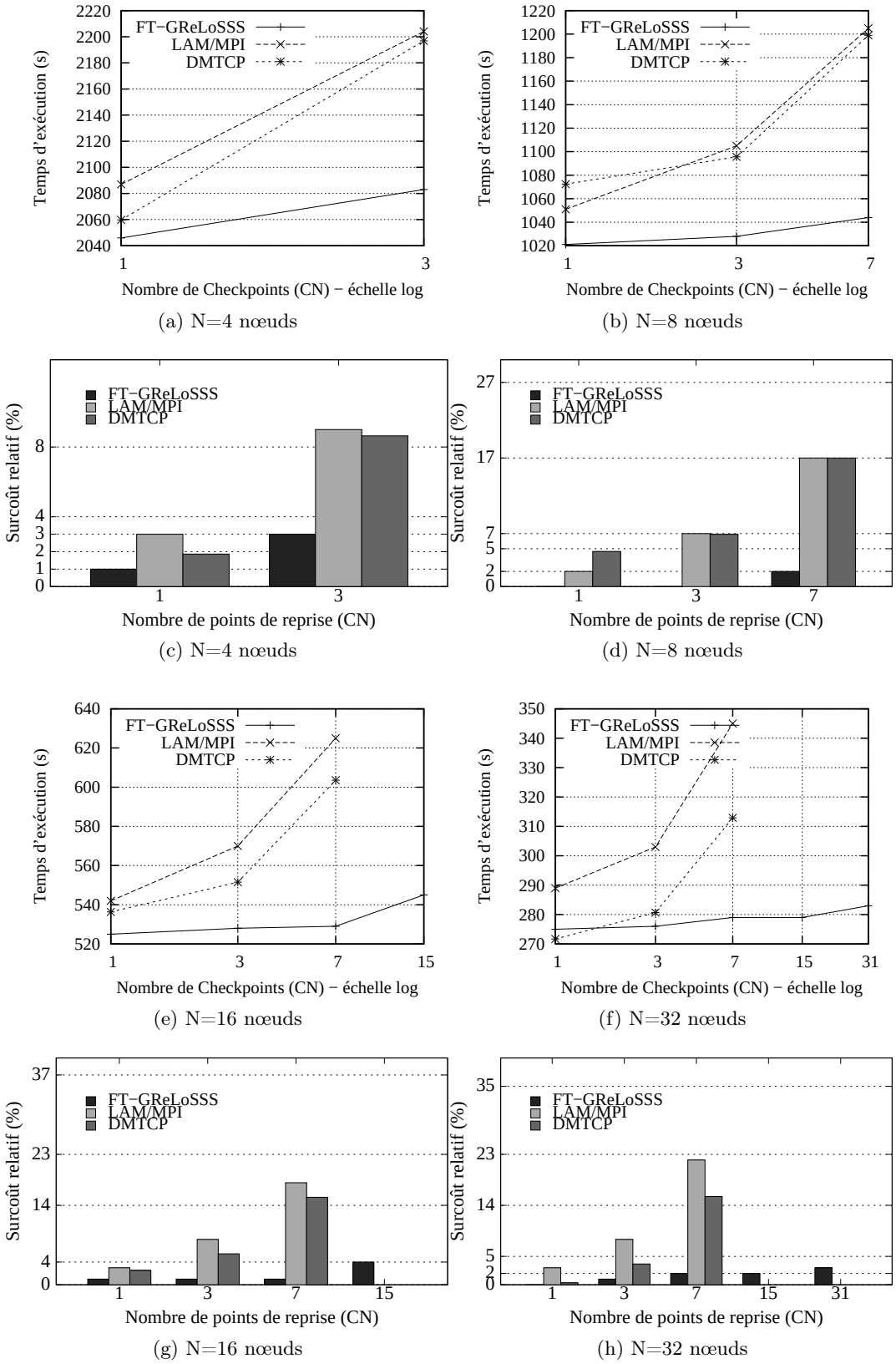
7.4.2 Coût comparatif de la tolérance aux pannes à l'exécution

Les graphiques des figures 7.4 et 7.5 sont consacrés à l'étude de l'impact du nombre de points de reprise réalisés sur les temps d'exécution respectivement pour des matrices de tailles 16384×16384 et 32768×32768 dans l'application *Matmult*. Chaque figure comporte des graphiques pour différents nombres de processus et compare les temps d'exécution affichés par FT-GReLoSSS, LAM/MPI et DMTCP. La comparaison des temps est effectuée en absolu. Nous fournissons également des diagrammes en bâtons représentant le surcoût relatif de chaque solution par rapport à son implantation MPI. Cette information a été jugée nécessaire pour une comparaison plus équitable, compte-tenu du fait que les temps de réference de LAM/MPI sont parfois plus longs que ceux d'OMPI (cf. Section 7.3).

Dans la plupart des cas, FT-GReLoSSS affiche les meilleurs résultats. Nous pouvons observer que pour un faible nombre de points de reprise réalisés — souvent compris entre 1 et 7 — les trois solutions affichent des temps relativement proches. En particulier, pour un unique point de reprise réalisé et de taille moyenne à faible (inférieure à 512 Mo), DMTCP est légèrement plus rapide (cf. Figure 7.4f). Ceci ne s'applique pas par exemple pour des matrices de taille 16384×16384 sur 4, 8 ou 16 nœuds (cf. Figures 7.4a, 7.4b et 7.4e) car les points de reprise sont alors volumineux [19] : ils sont au moins 50% plus volumineux pour LAM/MPI et DMTCP par rapport à FT-GReLoSSS de sorte que l'impact sur le temps d'exécution est plus important. Les surcoûts relatifs de LAM/MPI et DMTCP sont souvent très proches avec néanmoins un léger avantage pour DMTCP : probablement car ce dernier est plus récent.

Pour un nombre de points de reprise réalisés plus important — au-delà de 7 — l'écart se creuse de manière très importante en faveur de FT-GReLoSSS. Dans cette situation, deux facteurs interviennent. Le premier est la taille des points de reprise dont l'impact est très important

19. Compris entre 257 et 1126 Mo pour FT-GReLoSSS avec optimisations, entre 420 et 1638 Mo pour LAM/MPI, et entre 567 et 2102 Mo pour DMTCP (cf. Figure 7.3a)



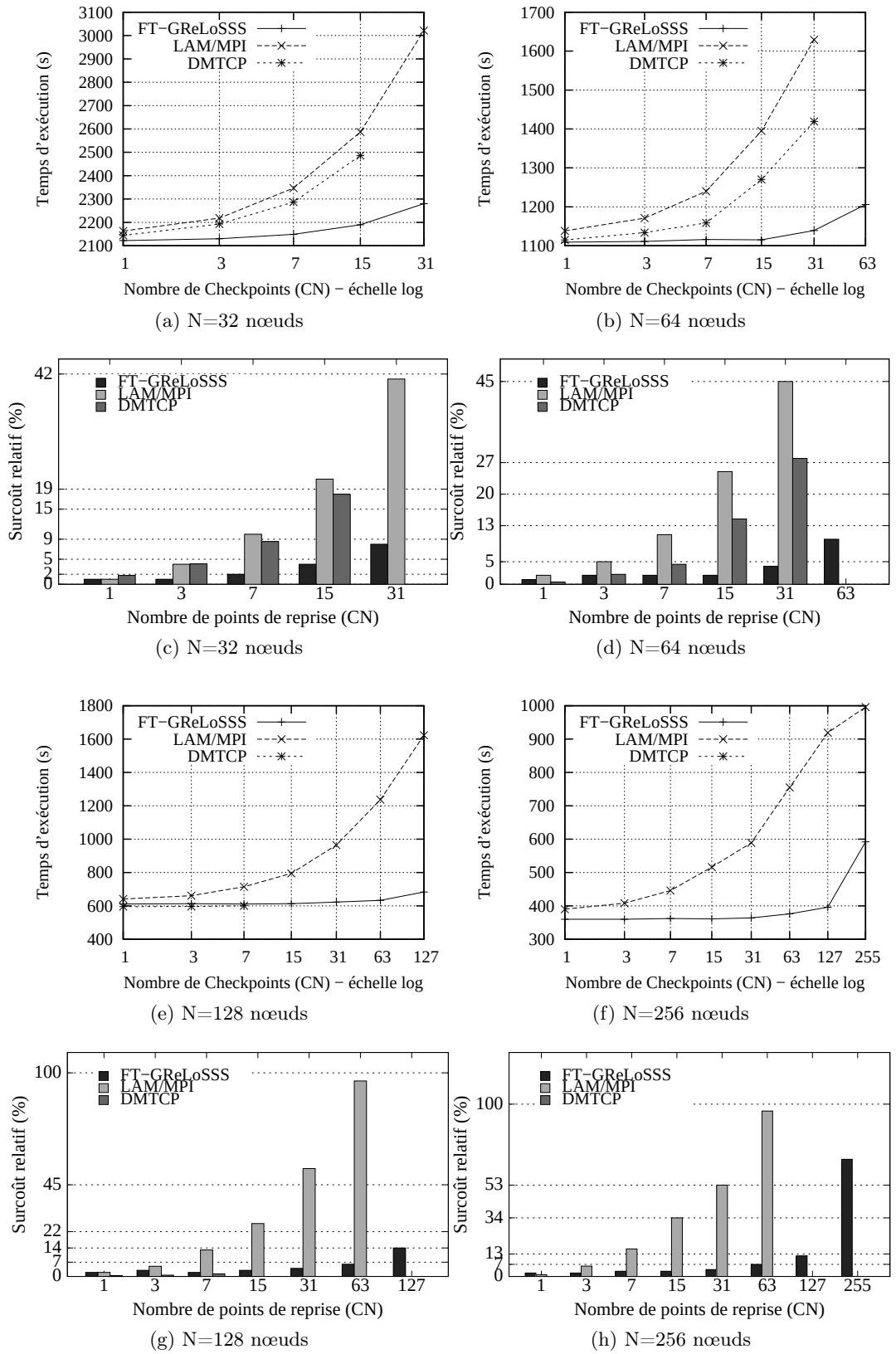
(a) N=4 nœuds (b) N=8 nœuds

(c) N=4 nœuds (d) N=8 nœuds

(e) N=16 nœuds (f) N=32 nœuds

(g) N=16 nœuds (h) N=32 nœuds

FIGURE 7.4 – Impact du nombre de points de reprise sur le temps d'exécution pour l'application *Matmult* avec des matrices de taille 16384 × 16384 et lorsque le nombre de nœuds N varie.



(a) N=32 nœuds (b) N=64 nœuds

(c) N=32 nœuds (d) N=64 nœuds

(e) N=128 nœuds (f) N=256 nœuds

(g) N=128 nœuds (h) N=256 nœuds

FIGURE 7.5 – Impact du nombre de points de reprise sur le temps d'exécution pour l'application *Matmult* avec des matrices de taille 32768 × 32768 et lorsque le nombre de nœuds N varie.

lorsque les volumes des points de reprise par processus sont importants. Une application comme *Matmult* qui admet une synchronisation au niveau des communications relativement forte peut souffrir en terme de performance si des désynchronisations sont créées. En effet, au moment des communications une synchronisation sur le(s) processus le(s) plus lent(s) sera réalisée avec pour effet de ralentir l'ensemble de l'application. L'écriture de points de reprise constitue une source de désynchronisation dont l'impact dépend (directement) de la taille des points de reprise : LAM/MPI et DMTCP sont donc désavantagés.

Cependant, la taille des points de reprise ne suffit pas pour expliquer tous les écarts observés. En effet, qu'en est-il des cas où l'augmentation du nombre de processeurs permet d'obtenir des points de reprise par processus de faible tailles ? C'est le cas par exemple pour des matrices de tailles 32768×32768 sur 256 nœuds (cf. Figures 7.5h et 7.5f) et où la taille des points de reprise pour DMTCP et LAM/MPI n'exède pas les 256 Mo (cf. Figure 7.3c). C'est dans ce type de situation qu'intervient le second facteur : le protocole de coordination. Contrairement au mécanisme de coordination de FT-GReLoSSS qui s'effectue sans aucune communication, ceux de LAM/MPI et DMTCP réalisent une synchronisation forte qui bloque les processus applicatifs (cf. Section 3.4.2, p. 33). Le temps de bloquage dépend du nombre de processus impliqués ainsi que du nombre de messages en transit pendant l'opération de coordination. En effet, le protocole attend qu'il n'y ait plus de messages sur le réseau pour enclencher l'écriture des points de reprise. En conséquence, le temps de bloquage s'additionne au temps requis pour écrire le point de reprise sur disque. En contre partie, avec FT-GReLoSSS, l'endroit de réalisation des points de reprise est beaucoup plus restreint que dans les SPRD de niveau systèmes : ces derniers peuvent réaliser des points de reprise à n'importe quel moment de l'exécution de l'application. Dans l'étude menée à la section 7.5.2 (p. 134) cette limitation sur le nombre de points de reprise qu'il est possible de réaliser s'est fait ressentir.

Enfin, il serait intéressant de se comparer à des SPRDs de niveau système intégrant d'autres protocoles de reprise par retour arrière et notamment des protocoles non bloquants. Actuellement, le seul que l'on connaisse et qui soit disponible est celui de MPICH-V [24] (cf. Section 3.7, p. 45). Il s'agit d'un projet du Laboratoire de Recherche en Informatique de l'université de Paris-Sud qui a integré de nombreux protocoles de coordination au sein de l'implantation MPICH de la spécification MPI (cf. [60] pour une présentation de MPICH). Plusieurs essais ont été effectués pour installer MPICH-V sur notre plate-forme d'expérimentation et pouvoir réaliser cette étude. Malheureusement, ces essais se sont averés infructueux et nous n'avons pu bénéficier d'aucune aide des développeurs étant donné l'arrêt du développement de MPICH-V.

Résultat. *Sur notre benchmark* Matmult*, FT-GReLoSSS a un ralentissement comparable à d'autres solutions de tolérance aux pannes pour un faible nombre de points de reprise réalisés ; et un ralentissement inférieur lorsque la fréquence de réalisation des points de reprise augmente.*

7.5 Performances de FT-GReLoSSS avec tolérance aux pannes en présence de pannes

Dans cette section nous procédons à l'évaluation des performances de la tolérance aux pannes de FT-GReLoSSS en présence de pannes. Pour ce faire, nous nous intéressons dans un premier temps au coût de la reprise sur panne et dans un second temps nous réalisons *une étude à surcoût constant*.

La figure 7.6 illustre une exécution (E1) d'une application associée à un SPRD mais en l'absence de panne ; et une exécution (E2) en présence d'une panne et suivie d'une reprise sur panne. Ceci donne lieu à deux exécutions notées (E2.1) et (E2.2). (E2.1) représente l'exécution initiale qui est interrompue par l'occurrence d'une panne, et (E2.2) représente l'exécution qui s'ensuit et pendant laquelle est réalisée la reprise sur panne. L'exécution (E2.2) comprend le temps mis pour détecter l'occurrence de la panne, ainsi que le temps nécessaire pour reconstituer en mémoire l'état de l'application au moment de la réalisation du point de reprise. Ces deux temps sont respectivement notés `T_detection` et `T_chargement` à la figure 7.6.

Les points de reprise dans la figure sont supposés définis dans le code source de l'application. Le principe reste le même pour le cas d'un SPRD de niveau système qui déclenche les points de reprise de manière externe à l'application. Suite à la détection de la panne qui peut être accomplie par l'utilisateur de l'application, par un mécanisme de détection automatique integré au SPRD ou par tout autre mécanisme propre à l'environnement d'exécution, l'application est relancée à partir de son dernier point de reprise. En fonction de l'instant d'occurrence de la panne par rapport à la réalisation du dernier point de reprise, le temps de calcul perdu sera plus ou moins important.

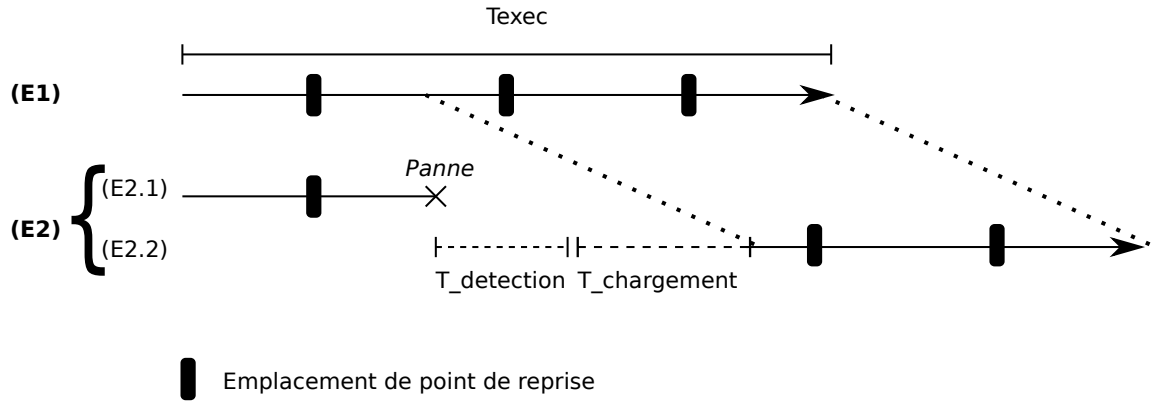


FIGURE 7.6 – Illustration des étapes de reprise sur pannes : cas d'un SPRD de niveau applicatif.

Nous allons maintenant étudier les performances de FT-GReLoSSS lorsque des pannes surviennent au cours de l'exécution de l'application *Matmult*. Les expériences n'incluent malheureusement pas les temps de reprises avec DMTCP. En effet, ce dernier n'arrivait pas à restaurer des points de reprise sur notre plate-forme d'expérimentation. Pourtant, des expériences d'autres configurations n'ont présenté aucun problème.

7.5.1 Coût comparatif de la reprise sur panne

7.5.1.1 Méthode de mesure du coût de la reprise sur panne

Le protocole de mesure du surcoût de la reprise sur panne est illustré à la figure 7.7. Pour mesurer le surcoût à l'exécution occasionné par une panne lorsqu'on utilise FT-GReLoSSS, nous exécutons l'application *Matmult* que nous réglons pour réaliser un point de reprise à mi-exécution, et s'interrompre immédiatement après. Ceci nous donne le temps $T1$ correspondant au temps qui s'est écoulé entre le début de l'exécution et l'occurrence de la panne. Ce temps comprend également le temps nécessaire à l'écriture du point de reprise sur disque. L'application est ensuite relancée à partir de son point de reprise. Nous mesurons alors $T2$ qui correspond au temps entre la relance de l'application et la fin de son exécution. Ce temps comprend le temps nécessaire

pour charger le point de reprise en mémoire à partir du disque (noté `T_chargement` à la figure 7.6). Pour une meilleure précision du temps mesuré, la reprise sur panne est iterée 5 fois et la valeur de $T2$ utilisée correspond à la valeur moyenne. La somme de ces temps (*i.e.* : $T1 + T2$) est comparée au temps T qui correspond au temps moyen d'exécution de l'application avec un point de reprise. Les temps T, $T1$ et $T2$ sont mesurés de manière externe à l'application avec la commande `UNIX` « `time` » (cf. Section 7.1.4, p. 117).

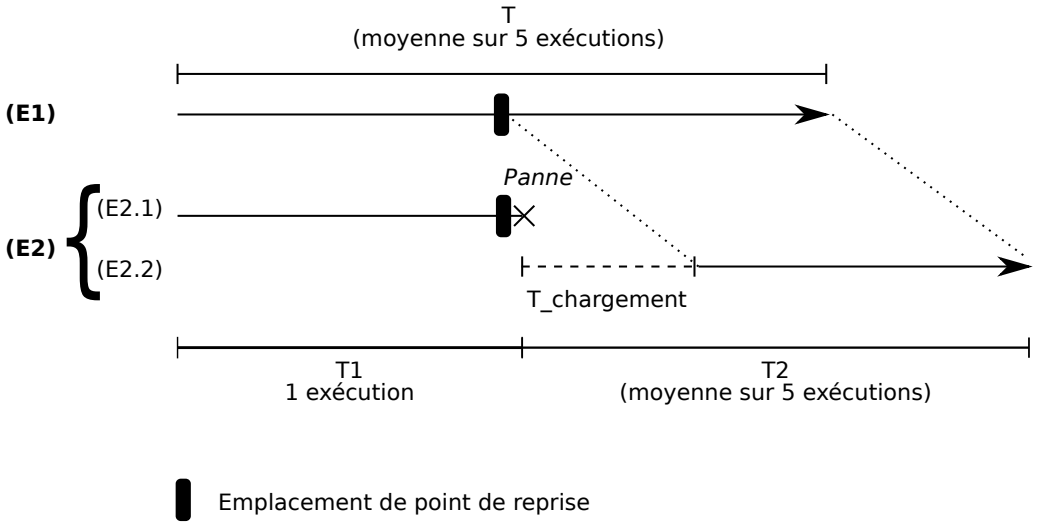


FIGURE 7.7 – Illustration du protocole de mesure du surcoût de reprise sur panne.

Pour LAM/MPI, la réalisation du point de reprise est également déclenchée environ à mi-exécution, mais de manière externe à l'application. Pour ce faire, nous utilisons la commande `lamcheckpoint` qui est fournie avec LAM/MPI. Au terme de l'exécution de cette commande, la réalisation du point de reprise est terminée ; nous provoquons aussitôt une panne de manière externe en mettant fin à l'exécution de l'application à l'aide de la commande `UNIX pkill`.

À la différence du cas plus général de la figure 7.6, dans cette expérience nous provoquons la panne immédiatement après la réalisation du point de reprise de manière à avoir un protocole d'expérimentation homogène entre FT-GReLoSSS et LAM/MPI : autrement il serait très difficile (voire impossible) de provoquer une panne au même moment.

7.5.1.2 Résultats de l'expérience sur l'application *Matmult*

Dans le cas de systèmes comme LAM/MPI, le temps nécessaire à une application pour reprendre son exécution suite à une panne peut être approximé par le temps nécessaire pour charger le point de reprise en mémoire. Pour FT-GReLoSSS, ce temps comprend en plus le temps pour déterminer la ligne de reprise et le temps pour restaurer le contexte d'exécution.

Les tableaux 7.8 et 7.9 reportent les résultats des différentes mesures réalisées pour FT-GReLoSSS et LAM/MPI respectivement. Les tableaux présentent les temps T et $T1+T2$ définis précédemment. Pour des matrices de tailles 16384×16384 les surcoûts liés à la reprise sont très faibles : inférieurs à 1.3% pour FT-GReLoSSS et inférieurs à 1.7% pour LAM/MPI. Pour des matrices de tailles 32768×32768 les surcoûts affichés par FT-GReLoSSS et LAM/MPI demeurent comparables et très faibles (inférieurs à 0.4%) sauf pour les exécutions sur 256 nœuds. LAM/MPI affiche un surcoût de 1.5% et FT-GReLoSSS un surcoût de 4.6%. Enfin pour des tailles de matrices de 65536×65536 les surcoûts pour FT-GReLoSSS sont compris entre 1.5% et 3.2%.

Ce sont des surcoûts qui sont plus élevés que pour les autres tailles de matrice mais qui restent faibles. Quant aux surcoûts de LAM/MPI ils sont tous négatifs : entre -1.9% et -0.6%. D'autres surcoûts négatifs peuvent également être observés pour les deux autres tailles de matrices : aussi bien pour FT-GReLoSSS que pour LAM/MPI. Ces surcoûts sont attribués d'une part à l'imprécision des mesures, d'autre part à la faible proportion du temps de chargement des points de reprise par rapport au temps total d'exécution de l'application, et enfin aux perturbations système. Ce résultat est similaire à celui des expérimentations sur l'évaluation de la performance de réalisation des points de reprise (cf. Section 7.4.2, p. 128) où la réalisation d'un unique point de reprise introduit un surcoût similaire pour les SPRD comparés. Les différences n'apparaissent que lorsque le nombre de points de reprise réalisés augmente. Ainsi, l'optimisation sur la taille des points de reprise n'est intéressante qu'à partir d'un certain nombre de points de reprise réalisés. Ce résultat devrait *a priori* rester valable pour la reprise : on devrait voir l'intérêt de points de reprise de taille réduite surtout pour un grand nombre de points de reprise.

Résultat. *Notre expérience sur l'application* Matmult *montre que FT-GReLoSSS et LAM/MPI admettent des coûts de reprise faibles et comparables mais un manque de précision des mesures ne permet pas de les départager.*

Taille des matrices	Nombre de nœuds	FT-GReLoSSS T_{exec} (s)		Surcoût absolu (s)	Surcoût relatif (%)
		$T \pm \sigma$	$T1 + T2$		
16384×16384	4	2048 ± 1.8	2057	9	0.4
	8	1024 ± 1.6	1037	13	1.3
	16	527 ± 0.5	531	3	0.6
	32	277 ± 0.4	280	3	0.9
32768×32768	32	2129 ± 5.1	2135	7	0.3
	64	1115 ± 2.2	1117	3	0.2
	128	618 ± 4.2	616	-2	-0.4
	256	375 ± 4.6	392	17	4.6
65536×65536	64	8503 ± 60.9	8682	178	2.1
	128	4595 ± 18.9	4662	67	1.5
	256	2471 ± 11.7	2550	79	3.2

TABLE 7.8 – Surcoût introduit par FT-GReLoSSS à la reprise sur panne pour l'application *Matmult* avec OMPI.

7.5.2 Comparatif de la tolérance aux pannes à surcoût constant

À la section 7.4.2 nous avons mis en évidence le meilleur comportement de FT-GReLoSSS par rapport à LAM/MPI et DMTCP, en particulier lorsque la fréquence de réalisation de points de reprise croît. Dans cette section, nous étendons l'étude en nous concentrant sur la quantification de la tolérance aux pannes pour un surcoût constant sur le temps d'exécution et fixé par l'utilisateur. L'objectif est de déterminer avec quel niveau de tolérance aux pannes LAM/MPI et FT-GReLoSSS arrivent à satisfaire cette contrainte sur le surcoût.

Taille des matrices	Nombre de nœuds	LAM/MPI T_{exec} (s)		Surcoût absolu (s)	Surcoût relatif (%)
		$T \pm \sigma$	$T1 + T2$		
16384×16384	4	2088 ± 15.6	2097	8	0.4
	8	1053 ± 3.9	1061	8	0.8
	16	544 ± 2.8	547	3	0.6
	32	291 ± 2.0	296	5	1.7
32768×32768	32	2168 ± 7.6	2160	-8	-0.4
	64	1145 ± 1.8	1147	2	0.2
	128	647 ± 2.6	650	3	0.4
	256	399 ± 1.5	405	6	1.5
65536×65536	64	8599 ± 23.0	8547	-52	-0.6
	128	4683 ± 35.3	4595	-87	-1.9
	256	2610 ± 18.7	2580	-31	-1.2

TABLE 7.9 – Surcoût introduit par LAM/MPI à la reprise sur panne pour l'application *Matmult*.

7.5.2.1 Description de l'expérience

L'étude est menée avec l'application *Matmult*, pour les trois tailles de matrices déjà utilisées dans les expériences précédentes et pour différents nombres de nœuds (toujours à raison d'un processus par nœud). Nous avons vu à la figure 7.6 que la quantité de travail (ou temps de calcul) perdue varie selon le moment d'occurrence de la panne : elle sera d'autant plus importante que le moment d'occurrence de la panne sera éloigné du moment de réalisation du dernier point de reprise.

Par la suite, nous choisissons de fixer le surcoût en temps d'exécution à 10%. Ce dernier correspond au surcoût desiré pour une exécution sans panne : il ne tient pas compte du surcoût introduit en cas de panne. Par ailleurs, nous supposons que les points de reprise sont réalisés à intervalles réguliers et constants. Pour chaque taille de matrice, nous déterminons le nombre de points de reprise que LAM/MPI et FT-GReLoSSS réussissent à réaliser en se conformant à notre limite de 10% (de surcoût en temps d'exécution). Comme nous aurons l'occasion de l'observer, FT-GReLoSSS ne pourra pas toujours aller jusqu'à cette limite étant donné qu'il est limité sur le nombre de points de reprise qu'il peut réaliser. Dans le cas de l'application *Matmult* la limite est le nombre de cycles de la boucle principale de l'application (cf. Section 5.4.2, p. 80), qui correspond au nombre de nœuds.

7.5.2.2 Résultats de l'expérience sur l'application *Matmult*

Les figures 7.8a et 7.8b reportent les résultats de l'expérience relatifs aux matrices de taille 16384×16384. La figure 7.8a trace le nombre maximal de points de reprise réalisés par LAM/MPI et FT-GReLoSSS en fonction du nombre de nœuds utilisés. Le surcoût relatif obtenu apparaît sous forme de pourcentage à proximité de chaque point tracé. La figure 7.8b trace le temps maximal perdu, et le nombre de points de reprise réalisés (noté *CN* pour *achieved Checkpoint Number*).

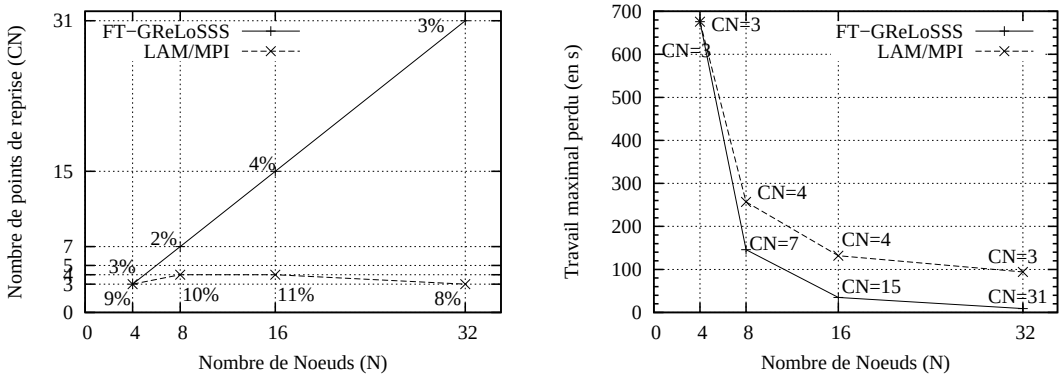
FT-GReLoSSS réalise pour chaque nombre de nœud, le nombre maximal de points de reprise qui lui est possible de réaliser sans dépasser un surcoût de 3%. LAM/MPI réussit à mieux profiter de la limite autorisée de +10% de temps d'exécution mais en réalisant au plus 4 points

de reprise dans tous les cas. Nous avons un exemple de situation dans laquelle FT-GReLoSSS ne peut réaliser davantage de points de reprise à cause des limitations imposées par la solution de tolérance aux pannes qu'il met en œuvre. LAM/MPI ne possède pas ce type de limitation, mais sa solution de tolérance aux pannes n'est pas suffisamment performante pour lui permettre d'obtenir un meilleur résultat.

Au final, pour un nombre de nœuds supérieur à 8, FT-GReLoSSS permet de perdre beaucoup moins de travail que LAM/MPI en cas de panne(s) (cf. Figure 7.8b).

Un comportement similaire s'observe pour des matrices de taille 32768×32768 (cf. Figures 7.9a et 7.9b) ainsi que pour des matrices de taille 65536×65536 (cf. Figures 7.10a et 7.10b) : le nombre de points de reprise que LAM/MPI arrive à réaliser – dans la limite des +10% de temps d'exécution – reste plus ou moins constant tandis que celui réalisé pas FT-GReLoSSS augmente avec le nombre de nœuds utilisés. Par rapport à l'expérience avec les matrices de taille 16384×16384, FT-GReLoSSS est moins limité par le nombre maximal de points de reprise qu'il peut réaliser. Mais finalement, FT-GReLoSSS est toujours plus efficace que LAM/MPI.

Résultat. *Grâce aux optimisations sur la taille des points de reprise et à son protocole de coordination sans communications, FT-GReLoSSS réussi à réaliser de nombreux points de reprise de manière à perdre moins de temps de travail que LAM/MPI sur notre benchmark* Matmult. *En revanche, l'existence d'un nombre maximal de points de reprise que FT-GReLoSSS permet de réaliser réduit les possibilités de réglage fin pour satisfaire une contrainte de l'utilisateur.*

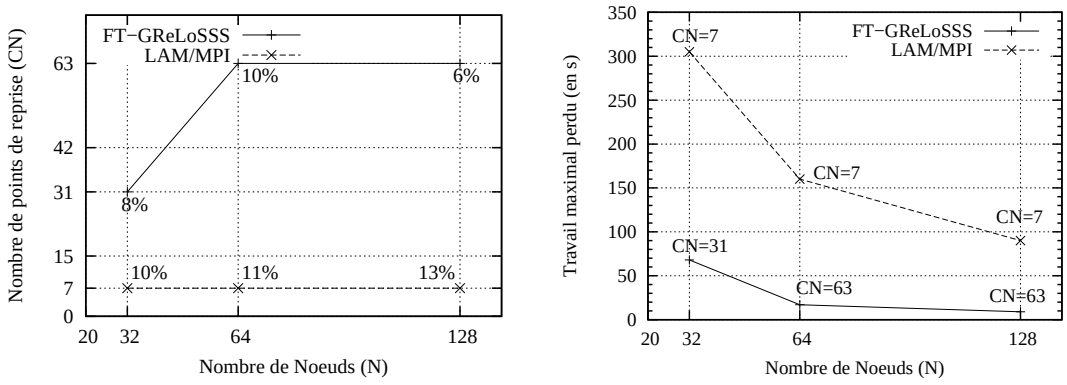


(a) (b)

FIGURE 7.8 – Étude à surcoût relatif presque constant (autour de 10%) pour le petit problème (16384×16384) de l'application *Matmult*.

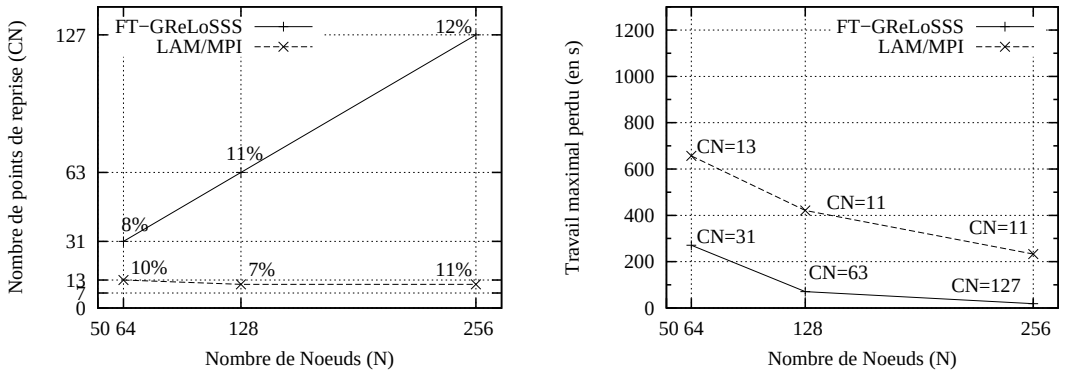
7.6 Évaluation des performances de ToMaWork sans tolérance aux pannes

À ce jour nous avons vérifié le bon fonctionnement de notre modèle et du framework ToMaWork mais nous n'avons pas encore eu le temps de l'évaluer quantitativement.



(a) (b)

FIGURE 7.9 – Étude à surcoût relatif presque constant (autour de 10%) pour le moyen problème (32768×32768) de l'application *Matmult*.



(a) (b)

FIGURE 7.10 – Étude à surcoût relatif presque constant (autour de 10%) pour le gros problème (65536×65536) de l'application *Matmult*.

Chapitre 8

Conclusion et perspectives

8.1 Bilan des travaux réalisés et des résultats obtenus

Au cours de cette thèse nous nous sommes intéressés à la recherche de solutions – efficaces et portables – pour tolérer les pannes d'applications de calculs distribués sur des grappes de PCs. Ces architectures distribuées deviennent de plus en plus répandues à cause de leur faible coût à l'achat, mais leur adoption pour des applications critiques, ou simplement soumises à des contraintes de temps, est freinée à cause de leur manque de fiabilité. Les travaux menés en début de thèse (**Chapitre 2**) nous ont confronté à cette réalité et ont constitué une source de motivation pour la suite de nos recherches.

Comme le montre notre état de l'art sur la tolérance aux pannes dans les grappes de PCs (**Chapitre 3**), les pannes qui peuvent survenir dans ces environnements complexes peuvent être de natures très variées, et provoquer des comportements différents. Un grand nombre d'entre elles sont cependant des *pannes par arrêt*, et résultent du passage à l'échelle des grappes (*i.e.* : augmentation du nombre de nœuds et de cœurs) mais aussi celui des applications qui s'y exécutent. Ce type de pannes auxquelles nous nous intéressons sont traditionnellement traitées par des solutions de reprise par retour arrière qui combinent des *protocoles de reprise par retour arrière* avec des *réalisations de points de reprise*. La mise en œuvre de ces solutions se réalise principalement au niveau système ou au niveau applicatif.

Dans notre approche, nous avons privilégié le niveau applicatif car il donne lieu à des réalisations de tolérance aux pannes très portables, et qui peuvent plus facilement bénéficier de la sémantique applicative pour gagner en efficacité. L'inconvénient majeur du niveau applicatif est de nécessiter la transformation du code source applicatif pour que l'application devienne capable de se sauvegarder pendant son exécution, et de reprendre son exécution à partir d'une sauvegarde.

Notre recherche à abouti à la définition du modèle de tolérance aux pannes MoLOToF (**Chapitre 4**). Ce dernier appelle à une organisation structurée des applications autour de *squelettes tolérants aux pannes* et à diverses *collaborations*, notamment entre le programmeur et le système de tolérance aux pannes. MoLOToF est un modèle qui dans sa forme actuelle contraint le programmeur, en particulier dans l'organisation du code de l'application en échange de quoi le code tolérant aux pannes demeure simple. Pour évaluer MoLOToF, nous avons appliqué ses principes à des familles d'algorithmes parallèles, et ce en utilisant des langages de programmation ainsi que des technologies de distribution différentes. Le choix d'appliquer les principes au niveau d'algorithmes parallèles plutôt qu'à un niveau plus bas, a été fait afin de faciliter la tâche du programmeur en ce qui concerne la tolérance aux pannes. En particulier, le programmeur n'a

plus à se soucier de la cohérence des points de reprise : le protocole de reprise par retour arrière lui est occulté.

FT-GReLoSSS (**Chapitre 5**) est le framework C++/MPI qui a résulté de l'application de MoLOToF à des algorithmes de décomposition de domaine qui suivent un modèle BSP relâché. Pour ces algorithmes, FT-GReLoSSS propose deux squelettes tolérants aux pannes permettant leur implantation. Ces squelettes possèdent la particularité d'intégrer une interface pour tableaux à N dimensions qui laisse plus de liberté au programmeur sur le choix de sa structure de données. Le plan de routage est défini aussi par le programmeur, et contribue également à lui laisser plus de liberté. L'utilisation de squelettes de programmation a permis de concevoir un nouveau protocole de niveau applicatif afin d'intéragir avec l'environnement d'exécution et de munir ainsi FT-GReLoSSS d'un *fonctionnement piloté de l'extérieur*.

L'évaluation de la facilité de développement de FT-GReLoSSS (**Chapitre 7**) montre que l'approche adoptée par FT-GReLoSSS est verbeuse et l'évaluation de FT-GReLoSSS sur certaines expériences en l'absence de tolérance aux pannes suggère quelques faiblesses au niveau du plan de routage. Cependant la verbosité de FT-GReLoSSS correspond, pour la plupart, à des opérations de niveau algorithmique (très) simple. Enfin, les expériences en présence de tolérance aux pannes démontrent que FT-GReLoSSS permet d'obtenir un meilleur niveau de tolérance aux pannes que des SPRDs de niveau système existants (LAM/MPI et DMTCP).

ToMaWork (**Chapitre 6**) est le framework Java/JavaSpaces qui a résulté de l'application de MoLOToF à des algorithmes Maître-Travailleur statique et dynamique au-dessus d'un système de mémoire partagée virtuelle. L'utilisation du langage Java avec ses capacités intrinsèques de sérialisation/désérialisation ont facilité la réalisation transparente et portable de points de reprise. L'utilisation des mécanismes de tolérance aux pannes de niveau intergiciel qui accompagnent la technologie JavaSpaces a permis de mettre en œuvre un protocole de tolérance aux pannes original fondé sur des transactions. Bien que le protocole actuel admette une limitation au niveau du respect total de l'atomicité de ses transactions, une solution qui ne remet pas en cause l'existant à été trouvée. Comparé à FT-GReLoSSS, ToMaWork réussit à atteindre un niveau de transparence plus élevé pour le programmeur, et aboutit à une tolérance aux pannes standard entièrement automatique. Le modèle MoLOToF n'a cependant pas encore été appliqué complètement à ToMaWork. Il semble notamment possible d'enrichir les squelettes de calcul pour faciliter l'insertion de points de reprise intermédiaires. Enfin, les squelettes auxquels nous avons abouti pour ToMaWork sont plus complexes que ceux de FT-GReLoSSS, et offrent difficilement la possibilité au programmeur d'y intervenir. Cependant, ils sont suffisamment flexibles pour convenir aux principales variantes du Maître-Travailleur.

8.2 Travaux futurs

Pour MoLOToF une piste intéressante d'extension serait l'étude de la faisabilité du support des contextes imbriqués : une application immédiate serait de disposer de la tolérance aux pannes au sein d'une méthode de calcul. Ceci requerrait notamment la capacité de transmettre le contexte d'une boucle (ou d'une fonction) à la boucle (ou à la fonction) imbriquée.

Par ailleurs, l'approche au niveau applicatif de MoLOToF lui confère une très grande portabilité et rend adaptée son utilisation dans des environnements hétérogènes, que nous serons sans doute amenés à côtoyer davantage dans le futur. Un exemple de ce type d'environnements sont les grappes munies de nœuds de calculs hybrides comportant des processeurs multicœurs et des processeurs graphiques. Dans ce cadre il serait donc intéressant d'étudier la manière d'appliquer

MoLOToF pour sauver l'état d'applications qui utilisent des CPUs et des GPUs pour réaliser leurs calculs.

Pour FT-GReLoSSS nous songeons à apporter des améliorations qui permettent de réduire sa verbosité et ainsi alléger la quantité de travail que doit fournir le programmeur pour obtenir de la tolérance aux pannes. Une autre amélioration consisterait à enrichir l'interface de collaboration de FT-GReLoSSS avec le programmeur de manière à réduire davantage la taille des données sauvées. Cette interface pourrait fournir une primitive qui dans le cadre de tableaux ne sauverait que les données utiles ou modifiées (*i.e.* : une sorte de point de reprise incrémental). Dans le cas du produit de matrices par exemple, le tableau des résultats se remplit au fur et à mesure des calculs. À chaque super-étape, il suffit de n'en sauver qu'une partie.

Par ailleurs, nous comptons implanter et évaluer le protocole que nous avons proposé pour doter FT-GReLoSSS d'un fonctionnement piloté, et ainsi faciliter les interactions avec l'environnement d'exécution. En particulier, pour recevoir directement des ordres de réalisation de points de reprise en cas de maintenance/panne, ou de changement de la fréquence de réalisation des points de reprises.

Enfin, FT-GReLoSSS a su démontrer ses qualités sur une application parallèle de produit de matrices. Il serait très intéressant d'essayer de l'utiliser pour d'autres applications. Une première application de calculs distribués est l'application financière du Swing Gazier que nous avons développée en début de thèse en collaboration avec EDF R&D (cf. Chapitre 2). Cette expérience serait très intéressante pour avoir un retour d'utilisation de FT-GReLoSSS dans une application réelle.

Pour ToMaWork nous sommes actuellement en train de réaliser une étude approfondie de ses performances. Les résultats que nous avons obtenus avec le protocole de tolérance aux pannes standard actuel sont encourageants. De plus, nous sommes en train d'implanter la solution que nous avons trouvée pour remédier à l'insuffisance du protocole actuel à respecter entièrement l'atomicité de ses transactions. Par ailleurs, certaines des améliorations de FT-GReLoSSS comme celle sur l'enrichissement des outils de collaboration entre le programmeur et le framework pourraient être envisagées pour ToMaWork. Autrement, un effort devra être apporté pour faciliter l'écriture du code qui rend possible la reprise sur panne à partir des points de reprise intermédiaires. Enfin, de manière similaire à FT-GReLoSSS, nous avons l'intention d'utiliser ToMaWork pour apporter de la tolérance aux pannes à une application Maître-Travailleur réelle et plus complexe : celle de l'algorithme AdaBoost utilisé dans diverses applications de *machine learning* à Supélec [53].

Enfin, une dernière perspective consisterait à intégrer les applications développées au-dessus des frameworks FT-GReLoSSS et ToMaWork dans un « écosystème » tolérant aux pannes, comme CIFTS. Le modèle de MoLOToF semble compatible avec les principes de ces écosystèmes. Mais il faudrait tester l'intégration de nos frameworks, et finir l'implantation et la validation de leur fonctionnement en mode piloté.

Annexe A

Analyse des performances de l'application distribuée de *Swing Gazier*

Dans cette annexe nous détaillons les résultats des expériences réalisées en vue d'évaluer les performances de l'application distribuée de *Swing Gazier* que nous avons présentée au chapitre 2. Les expériences réalisées impliquaient trois modèles de prix du gaz aux besoins variés (cf. Section 2.3.1), ainsi que trois architectures distribuées différentes qui incluaient des grappes de PCs et le supercalculateur Blue Gene/L d'EDF R&D (cf. Section 2.3.2). Pour la grappe de bi-Opteron et le supercalculateur Blue Gene deux processeurs étaient présents par nœud. Il était donc possible d'exécuter l'application en utilisant une configuration avec un processeur par nœud (*mode monopro*) ou en utilisant une configuration avec deux processeurs par nœud (*mode bipro*). Les deux configurations ont été étudiées.

Les expériences ont consisté en la mesure des temps d'exécution de l'application pour chaque modèle de prix de gaz sous les différentes architectures distribuées considérées. Les résultats et leurs analyses sont présentés ci-après pour chaque modèle de prix de gaz considéré.

A.1 Modèle gaussien à un facteur

Sur ce modèle, le supercalculateur Blue Gene se comporte aussi bien dans les deux modes. Ce n'est pas le cas de la grappe de bi-Opteron dont les performances se dégradent considérablement en mode bipro à partir de 32 processeurs. Des mesures expérimentales ont montré que l'augmentation du nombre de processeurs accentuait de manière significative le temps passé sur les communications en mode bipro par rapport au mode monopro : il semblerait que l'accès à la carte réseau constitue une source d'engorgement pour les processeurs d'un même nœud. La suite se concentre sur l'analyse des performances en mode monopro présentes à la figure A.1.

L'observation des courbes d'accélérations de la figure A.1 rend compte d'une hyperaccélération entre 4 et 64 processeurs pour le supercalculateur Blue Gene et pour la grappe de bi-Opteron que nous attribuons à une augmentation simultanée du nombre de processeurs et de la quantité totale de mémoire disponible. Sachant que les nœuds sur Blue Gene disposent de deux fois moins de mémoire que les nœuds de la grappe de bi-Opteron, il n'est pas surprenant de voir ce dernier hyperaccélérer davantage. Dans tous les cas, l'hyperaccélération tend à disparaître avec l'augmentation du nombre de processeurs et l'accélération maximale est obtenue par Blue Gene sur 512 processeurs. La grappe de Pentium 4 ne réalise aucune hyperaccélération et sa courbe

d'accélération croît lentement. Il s'avère donc qu'un réseau d'interconnexions très rapide est indispensable pour obtenir de bonnes performances sur ce modèle, mais dès lors une grappe de taille moyenne telle que la grappe de bi-Opteron suffit pour en accélérer l'exécution.

En dernier lieu, malgré quelques problèmes de passage à l'échelle que nous avons rencontrés, le meilleur temps séquentiel obtenu, proche de 15 min, a pu être réduit à $13 - 15$ s sur une grappe de PCs et sur un supercalculateur Blue Gene. Il s'agit donc d'une réelle amélioration pour les utilisateurs étant donné que c'est le modèle le plus utilisé.

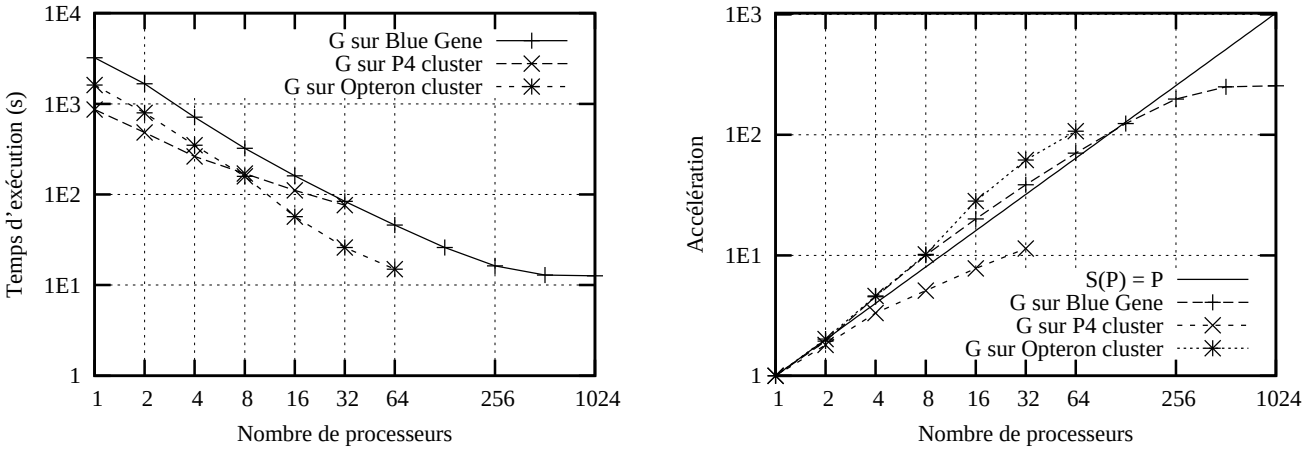


FIGURE A.1 – Temps d'exécution et accélérations (échelle logarithmique) avec le modèle gaussien sur trois architectures distribuées différentes en utilisant un seul processeur par nœud.

A.2 Modèle normal inverse gaussien

Que ce soit en mode monopro ou bipro les performances obtenues par notre algorithme et son implantation sur le modèle normal inverse gaussien sont très bonnes. Sur la figure A.2 où ont été reportées les performances mesurées en utilisant le nombre maximal de processeurs par nœuds (i.e. : mode bipro), nous observons une parallélisation quasi-parfaite et ce, même sur la grappe bas de gamme à base de Pentium 4. Le meilleur temps d'exécution est réalisé sur Blue Gene avec 1024 processeurs. Cependant, la grappe de bi-Opteron réalise une performance similaire avec seulement 128 processeurs. Ainsi, une alternative intéressante pour exécuter le modèle normal inverse gaussien avec notre algorithme est une grappe de PCs de grande taille munie de processeurs puissants sans être nécessairement reliés par un réseau ultra rapide.

Finalement, le meilleur temps séquentiel qui est proche de $6h25$ et qui a été obtenu par un processeur Opteron, a pu être réduit à 3 min en utilisant 1024 processeurs de Blue Gene. Il en résulte que notre distribution, moyennant la disponibilité d'un nombre suffisant de processeurs, rend possible une utilisation beaucoup plus courante du modèle normal inverse gaussien.

A.3 Modèle gaussien à deux facteurs

Ce modèle nécessite une puissance de calcul et un espace mémoire considérables en vue de s'exécuter dans des temps raisonnables (ou acceptables). Avec le jeu de paramètres courant et notre implantation qui utilise principalement deux tables - pour stocker les anciens et les nouveaux résultats - l'application aurait théoriquement besoin de 2×5895 Mo de memoire pour

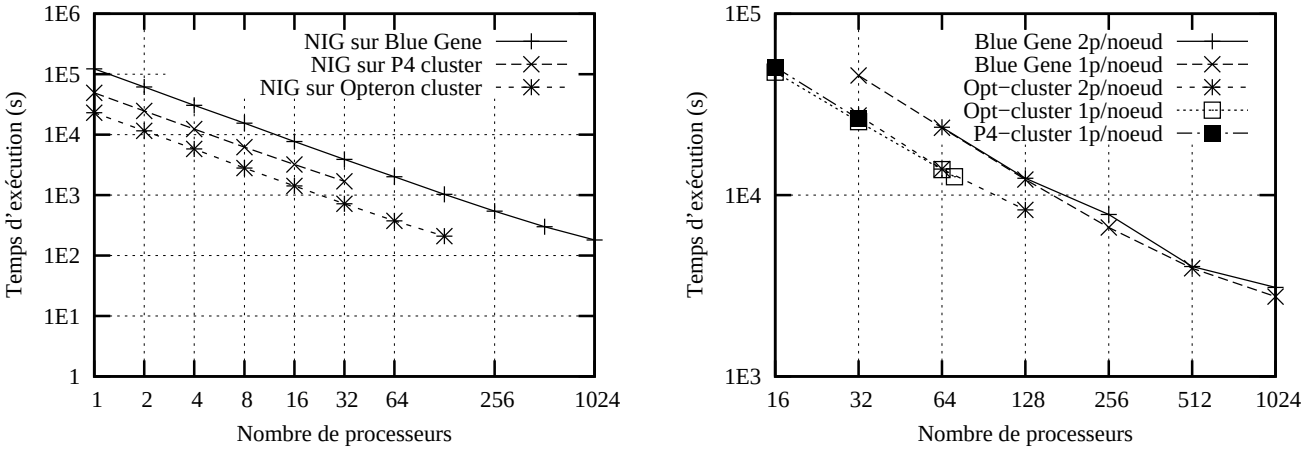


FIGURE A.2 – Temps d'exécution (échelle logarithmique) sur trois architectures distribuées différentes. À gauche : avec le *modèle normal inverse gaussien* en utilisant le nombre maximal de processeurs par nœud. À droite : avec le *modèle gaussien à deux facteurs* en utilisant le nombre minimal ou maximal de processeurs par nœud.

s'exécuter en séquentiel. En théorie, une distribution sur 8 nœuds nécessiterait 1474 Mo (*i.e.* : $(2 \times 5895)/8$) de RAM par nœud, donc 8 nœuds munis de 2 *Go* de RAM devraient supporter cette application. Cependant, en pratique, d'autres paramètres entrent en considération. Par exemple, la taille du noyau du système d'exploitation ne peut être négligée. Par ailleurs, à cause de la structuration de notre algorithme parallèle, la quantité de mémoire requise lors de la parallélisation est plus importante que dans le cas séquentiel. En l'occurrence, comme chaque processeur stocke les données influençant ses calculs, une légère réplication des données est possible au niveau des frontières de calculs entre processeurs. Il en résulte, qu'en pratique, cet algorithme nécessite au moins 10 processeurs dotés de 2 *Go* de mémoire chacun pour s'exécuter sans avoir recours au swap. Ainsi, selon le système, et sachant que l'on effectue nos tests avec des nombres de processeurs qui sont une puissance de 2, les courbes des temps d'exécution de la figure A.2 débutent soit à 16 soit à 32 processeurs. La petite grappe de Pentium-4 avec ses 32 nœuds monoprocesseurs équipés de 2 *Go* de mémoire a réussi à exécuter ce benchmark sur 16 et 32 processeurs. Avec une réduction du temps sur 32 processeurs d'un facteur deux par rapport au temps sur 16 processeurs, l'application semble réaliser un début de passage à l'échelle sur cette modeste grappe. La grappe de bi-Opteron, avec 2 *Go* de memoire par nœud, réussi également à exécuter le modèle Gaussien à deux facteurs à partir de 16 nœuds en mode monopro. Les temps d'exécution affichés en mode bipro sont très similaires à ceux en mode monopro et le benchmark supporte très bien le passage à l'échelle jusqu'à 128 processeurs sur 64 nœuds et en utilisant deux processeurs par nœud. En ce qui concerne le supercalculateur Blue Gene, avec seulement 1 *Go* de mémoire par nœud, la mémoire de 32 nœuds est nécessaire. Les temps en mode bipro sont légèrement plus importants que ceux en mode monopro mais le ralentissement observé est relativement faible. Comparé aux grappes de PCs, le supercalculateur Blue Gene qui est doté de la configuration par nœud la moins puissante connaît des temps d'exécution sur 32 processeurs beaucoup plus importants. Cependant grâce à son architecture *ultrascalable* [35] qui allie un excellent réseau d'interconnexions avec de nombreux nœuds, le supercalculateur Blue Gene démontre sa supériorité en finissant loin en tête.

Annexe B

Liste des publications

Les travaux effectués dans le cadre de cette thèse ont donné lieu à certaines publications que nous énumérons ci-après.

Les travaux sur la distribution de l'algorithme de contrôle stochastique réalisé en collaborations avec EDF R&D a donné au lieu aux publications suivantes :

- C. Makassikis, X. Warin and S. Vialle. Distribution of a Stochastic Control Algorithm Applied to Gas Storage Valuation. *The 7^{th} IEEE International Symposium on Signal Processing and Information Technology (ISSPIT '07), 2007.*
- C. Makassikis, X. Warin and S. Vialle. Large Scale Distribution of Stochastic Control Algorithms for Financial Applications. *The First Workshop on Parallel and Distributed Computing in Finance (Computational Finance) (PDCoF08), 2008.*
- C. Makassikis. Distribution Large Échelle d'un Algorithme Financier de Contrôle Stochastique. *$18^{èmes}$ Rencontres Francophones du Parallélisme, 2008.*

Les travaux et les résultats sur le modèle MoLOToF et le framework FT-GReLoSSS ont fait l'objet d'une publication à la conférence PDCAT 2010 :

- C. Makassikis, V. Galtier and S. Vialle. A Skeletal-Based Approach for the Development of Fault-Tolerant SPMD Applications. 11^{th} *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010.*

Les travaux sur le framework ToMaWork ont fait l'objet d'une publication à la conférence PDP 2011 :

- V. Galtier, C. Makassikis and S. Vialle. A Framework for Efficient Fault-tolerant Master-Worker Distributed Applications. 19^{th} *Euromicro International Conference on Parallel, Distributed and Network-Based (PDP), 2011.*

Bibliographie

[1] A. Agbaria and R. Friedman. Starfish : Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. *Cluster Computing*, 6(3) :227–236, 2003.

[2] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. Cactus Tools for Grid Applications. *Cluster Computing*, 4(3) :179–188, 2001.

[3] L. Alvisi, S. Rao, S. A. Husain, A. de Mel, and E. Elnozahy. An Analysis of Communication-Induced Checkpointing. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, Washington, DC, USA, 1999. IEEE Computer Society.

[4] J. Ansel, K. Aryay, and G. Cooperman. DMTCP : Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[5] J. N. C. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. DOME : Parallel programming in heterogeneous multi-user environment. Technical Report CMU-CS-95-137, Carnegie Mellon University, apr 1995.

[6] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HOTOS)*. IEEE Computer Society, 2001.

[7] A. Avizienis and J.-C. Laprie. Dependable computing : From concepts to design diversity. In *Proceedings of the IEEE*, pages 629–638, may 1986.

[8] A. Avizienis, J.-C. Laprie, and B. Randell. Dependability and Its Threats - A Taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, 2004.

[9] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Gener. Comput. Syst.*, 13(4-5) :361–372, 1998.

[10] C. Barrera-Esteve, F. Bergeret, C. Dossal, E. Gobet, A. Meziou, R. Munos, and D. Reboul-Salze. Numerical methods for the pricing of Swing options : a stochastic control approach. *Methodology and Computing in Applied Probability*, 8(4) :517–540, 2006.

[11] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2) :147–155, 1997.

[12] Beowulf FAQ. http://beowulf.org/overview/faq.html.

[13] The Blitz++ library. http://www.oonumerics.org/blitz/.

[14] The Boost C++ libraries. http://www.boost.org/.

[15] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V : Toward a

Scalable Fault Tolerant MPI for Volatile Nodes. In *Proceedings of the 15th ACM/IEEE Supercomputing Conference (SC)*. IEEE Computer Society Press, 2002.

[16] A. Bouteiller, F. Cappello, T. Herault, K. Krawezik, P. Lemarinier, F. Pierre, and F. Magniette. MPICH-V2 : a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender-Based Message Logging. In *Proceedings of the 16th ACM/IEEE Supercomputing Conference (SC)*, 2003.

[17] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V : a Multiprotocol Fault Tolerant MPI. *International Journal of High Performance Computing and Applications*, 20(3) :319–333, 2006.

[18] G. Bronevetsky. *Portable checkpointing for parallel applications*. PhD thesis, Cornell University, Department of Computer Science, 2007.

[19] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Recent Advances in Checkpoint/Recovery Systems. *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[20] G. Bronevetsky, D. Marques, K. Pingali, and R. Rugina. Compiler-Enhanced Incremental Checkpointing Applications. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2007.

[21] G. Bronevetsky, D. Marques, K. Pingali, R. Rugina, and S. A. McKee. Compiler-Enhanced Incremental Checkpointing for OpenMP Applications. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, may 2009.

[22] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated Application-level Checkpointing of MPI Programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 84–94, New York, NY, USA, 2003. ACM Press.

[23] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective Operations in Application-level Fault-tolerant MPI. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, pages 234–243, New York, NY, USA, 2003. ACM Press.

[24] D. Buntinas, C. Coti, T. Hérault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols. *Future Generation Computer Systems*, 24(1) :73–84, 2008.

[25] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 1. Addison Wesley Longmain, 3 edition, mar 2001.

[26] G. Burns, R. Daoud, and J. Vaigl. LAM : An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[27] R. Buyya. *High Performance Cluster Computing : Programming and Applications*, volume 2, chapter 1, pages 4–27. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[28] C. Makassikis and X. Warin and S. Vialle. Distribution of a Stochastic Control Algorithm Applied to Gas Storage Valuation. *The 7th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, 2007.

[29] C. Mishra and M.R. Mittal and S.K. Aggarwal. Checkpointing Fortran/MPI Programs for Computational Grids. In *Advances in Computer Science and Technology*, 2004.

[30] Cactus code. http://www.cactuscode.org/.

[31] B. Celic. Fault tolerance techniques for distributed systems, 2004. IBM technical article on developerWorks Rational library, available at http://www.ibm.com/developerworks/rational/library/114.html.

[32] S. Chakravorty and L. V. Kale. A Fault Tolerant Protocol for Massively Parallel Machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[33] S. Chakravorty and L. V. Kale. A Fault Tolerance Protocol with Fast Fault Recovery. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

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

[35] S. Chari. Breakthrough Advantage in Computational Fluid Dynamics with the IBM System Blue Gene Solution. *Vetrei Inc. White Paper*, sep 2006.

[36] Y. Chen, J. Plank, and K. Li. CLIP : A Checkpointing Tool for Message-Passing Parallel Programs. In *Proceedings of the 10th ACM/IEEE Supercomputing Conference (SC)*, 1997.

[37] J. O. Coplien. Curiously recurring template patterns. *C++ Report (SIGS Publications)*, 7(2), 1995.

[38] Contrat de Projet État-Région MIS. http://www.loria.fr/~falex/index.php/Adm/CperMis.

[39] F. Cristian, H. Aghali, and R. Strong. Clock Synchronization in the Presence of Omission and Performance Failures, and Processor Joins. In *Proceedings of the 16th International Symposium on Fault-Tolerant Computing Systems (FTCS)*. IEEE Computer Society Press, 1992.

[40] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic Broadcast : From Simple Message Diffusion to Byzantine Agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE Computer Society Press, 1985.

[41] F. Cristian and F. Jahanian. A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, 1991.

[42] DMTCP : Distributed MultiThreaded CheckPointing. http://dmtcp.sourceforge.net.

[43] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Comput. Surv.*, 34(3) :375–408, 2002.

[44] E. N. Elnozahy, D. B. Johnson, and W. Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.

[45] C. Engelmann, S. L. Scott, C. B. Leangsuksun, and X. B. He. Symmetric Active/Active High Availability for High-Performance Computing System Services. *Journal of Computers (JCP)*, 1(8) :43–54, 2006.

[46] C. Engelmann, G. R. Vallée, T. Naughton, and S. L. Scott. Proactive Fault Tolerance Using Preemptive Migration. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP)*, pages 252–257. IEEE Computer Society, Los Alamitos, CA, USA, 2009.

[47] G. E. Fagg and J. J. Dongarra. FT-MPI : Fault Tolerant MPI, supporting dynamic applications in a dynamic world. *Lecture Notes in Computer Science*, 1908 :346–353, 2000.

[48] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. Dongarra. Process Fault Tolerance : Semantics, Design and Applications for High Performance Computing. *International Journal of High Performance Computing Applications (IJHPCA)*, 19(4) :465–477, 2005.

[49] J. Fernández, E. Frachtenberg, and F. Petrini. BCS-MPI : A New Approach in the System Software Design for Large-Scale Parallel Computers. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing*. ACM, 2003.

[50] E. Frachtenberg, K. Davis, F. Petrini, J. Fernández, and J. C. Sancho. Designing Parallel Operating Systems via Parallel Programming. In *Euro-Par*, pages 689–696, 2004.

[51] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. 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, sep 2004.

[52] E. Gabriel, G. E. Fagg, A. Bukovsky, T. Angskun, and J. Dongarra. A Fault-Tolerant Communication Library for Grid Environments. In *Proceeding of the 17th Annual ACM International Conference on Supercomputing (ICS), International Workshop on Grid Computing and e-Science*, 2003.

[53] V. Galtier, S. Genaud, and S. Vialle. Implementation of the AdaBoost Algorithm for Large Scale Distributed Environments : Comparing JavaSpace and MPJ. In *15th International Conference on Parallel and Distributed Systems (ICPADS)*, dec 2009.

[54] R. Garg, V. K. Garg, and Y. Sabharwal. Scalable Algorithms for Global Snapshots in Distributed Systems. In *Proceedings of the 20th International Conference on Supercomputing (ICS)*, pages 269–277, 2006.

[55] A. H. Gebremedhin, I. G. Lassous, J. Gustedt, and J. A. Telle. PRO : A Model for Parallel Resource-Optimal Computation. In *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS)*. IEEE Computer Society, 2002.

[56] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1) :80–112, 1985.

[57] B. Goetz. Java theory and practice : Generics gotchas – Identify and avoid some of the pitfalls in learning to use generics, jan 2005. http://www.ibm.com/developerworks/java/library/j-jtp01255.html.

[58] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus Framework and Toolkit : Design and Applications. In *High Performance Computing for Computational Science – 5th International Conference on Vector and Parallel Processing (VECPAR)*. Springer-Verlag, 2002.

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

[60] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6) :789–828, sep 1996.

[61] R. Gupta, P. Beckman, H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra. CIFTS : A Coordinated infrastructure for Fault-Tolerant Systems. In *Proceedings of the 38th International Conference on Parallel Processing (ICPP)*, 2009.

[62] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31 :532–533, 1988.

[63] C. Huang. Thesis : System Support for Checkpoint/Restart of Charm++ and AMPI Applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[64] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 306–322, oct 2003.

[65] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software Rejuvenation : Analysis, Module and Applications. In *25th Symposium on Fault Tolerant Computer Systems*, pages 381–390, 1995.

[66] Y. Huang and Y.-M. Wang. Why Optimistic Message Logging Has Not Been Used in Telecommunications Systems. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS)*, 1995.

[67] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[68] H.W. Meuer and E. Strohmaier and J. J. Dongarra and H. D. Simon. TOP500 Supercomputing Sites. The report can be downloaded from : http://www.top500.org/, jun 2007. 29th Edition.

[69] Projet Intercell. http://intercell.metz.supelec.fr/spip.php?rubrique13.

[70] J. Dollimore and T. Kindberg and G. Coulouris. *Distributed Systems : Concepts and Design*. Pearson Education, 2005. Fourth Edition.

[71] P. Jaillet, E. I. Ronn, and S. Tompaidis. Valuation of Commodity-Based Swing Options. *Manage. Sci.*, 50(7) :909–921, 2004.

[72] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing (FTCS)*. IEEE Computer Society Press, 1987.

[73] L. Kalé and S. Krishnan. CHARM++ : A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, sep 1993.

[74] Kerrighed – Linux clusters made easy. http://www.kerrighed.org/.

[75] G. Kola, T. Kosar, and M. Livny. Faults in Large Distributed Systems and What We Can Do About Them. In *Proceeding of the 11th International Euro-Par Conference*, pages 442–453. Springer-Verlag, 2005.

[76] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *Software Engineering, IEEE Transactions on*, SE-13(1) :23–31, 1987.

[77] O. Koren. A study of the Linux kernel evolution. *SIGOPS Operating Systems Review*, 40(2) :110–112, 2006.

[78] H. Kuchen. A Skeleton Library. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, London, UK, 2002. Springer-Verlag.

[79] LAM/MPI Parallel Computing. http://www.lam-mpi.org.

[80] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3) :382–401, 1982.

[81] C. B. Leangsuksun, L. Shen, T. Liu, and S. L. Scott. Availability Prediction and Modeling of High Availability OSCAR Cluster. In *Proceedings of IEEE Cluster Computing*, 2003.

[82] S.-I. Lee, T. A. Johnson, and R. Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *Proceedings of 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, jun 2003.

[83] P. Lemarinier, A. Bouteiller, T. Hérault, G. Krawezik, and F. Cappello. Improved Message logging versus Improved coordinated checkpointing for fault tolerant MPI. In *CLUSTER*, pages 115–124, 2004.

[84] Librato Availability Services. http://www.evergrid.com/products/availability_services.

[85] C. Makassikis. Distribution Large Échelle d'un Algorithme Financier de Contrôle Stochastique. In *Proceedings des 18èmes Rencontres Francophones du Parallélisme (RenPar'18)*, 2008.

[86] D. Marques. *Automatic application-level checkpointing for high-performance computing systems*. PhD thesis, Cornell University, Department of Computer Science, 2006.

[87] F. Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation . *Journal of Parallel and Distributed Computing*, 18(4) :423–434, 1993.

[88] J. Mehnert-Spahn, E. Feller, and M. Schöttner. Incremental Checkpointing for Grids. In *Proceedings of Linux Symposium*, 2009.

[89] Message Passing Interface Forum. *MPI : A Message-Passing Interface Standard, Version 2.2*, sep 2009. http://www.mpi-forum.org/docs/docs.html.

[90] C. Mishra, M. R. Mital, and S. K. Aggarwal. Checkpointing Fortran/MPI programs for Computational Grids. *In Advances in Computer Science and Technology*, 2004.

[91] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), apr 1965.

[92] G. E. Moore. Progress In Digital Integrated Electronics. IEEE Text Speech, 1975.

[93] J. E. Moreira and V. K. Naik. Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications. Technical Report Research Report RC 20890, IBM, 1997.

[94] J. Morrison. The ASCI Q System at Los Alamos, mar 2003. 7th Workshop on Distributed Supercomputing (SOS7).

[95] MPICH2 : High-performance and Widely Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/.

[96] G. L. Mullen-Schultz and C. Soza. IBM System Blue Gene Solution : Application Development, 2005. IBM Redbook, Ref. ZG24-6745-00.

[97] V. K. Naik, S. P. Midkiff, and J. E. Moreira. A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems. In *Proceedings of the 10th ACM/IEEE Supercomputing Conference (SC)*, New York, NY, USA, 1997. ACM.

[98] V. P. Nelson. Fault-Tolerant Computing : Fundamental Concepts. *Computer*, 23(7) :19–25, 1990.

[99] The NetCDF Documentation. http://www.unidata.ucar.edu/software/netcdf/docs/.

[100] N. Neves and W. K. Fuchs. Using Time to Improve the Performance of Coordinated Checkpointing. In *Proceedings of the 2nd International Computer Performance and Dependability Symposium (IPDS '96)*, 1996.

[101] N. Neves and W. K. Fuchs. Coordinated Checkpointing without Direct Coordination. In *Proceedings of IEEE International Computer Performance & Dependability Symposium*, pages 23–31, 1998.

[102] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm. A SLOC Counting Standard. Technical Report TR-737, University of Southern California Viterbi – Center for Systems and Software Engineering, 2007.

[103] A. Nguyen-Tuong. *Integrating Fault-Tolerance Techniques in Grid Applications*. PhD thesis, University of Virginia, 2000.

[104] A. Oliner, L. Rudolph, and R. Sahoo. Cooperative checkpointing theory. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[105] A. J. Oliner and R. Sahoo. Evaluating cooperative checkpointing for supercomputing systems. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[106] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance Implications of Periodic Checkpointing on Large-Scale Cluster Systems. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS) - Workshop 18*. IEEE Computer Society, 2005.

[107] Open MPI : A High Performance Message Passing Library. http://www.open-mpi.org/.

[108] Oracle. *Java RMI Specification*, 2010. v. 1.6, ch. 7.

[109] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall PTR ; US Edition, 1994.

[110] A. Parker. Into the Wide Blue Yonder with BlueGene/L, apr 2005. Appeared in Science and Technology Review (UCRL-TR-52000-05-4), Lawrence Livermore National Laboratory, available at https://www.llnl.gov/str/April05/Seager.html.

[111] F. Petrini, K. Davis, and J. C. Sancho. System-Level Fault-Tolerance in Large-Scale Parallel Machines with Buffered Coscheduling. In *Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[112] J. S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Princeton University, jun 1993.

[113] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, University of Tennessee, jul 1997.

[114] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt : Transparent Checkpointing under Unix. In *Proceedings of USENIX Winter1995 Technical Conference*, pages 213–224, New Orleans, Louisiana/U.S.A., 1995.

[115] J. S. Plank and K. Li. Ickp : A Consistent Checkpointer for Multicomputers. *IEEE Parallel & Distributed Technology : Systems & Applications*, 2(2) :62–67, 1994.

[116] J. S. Plank, J. Xu, and R. Netzer. Compressed differences : An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, 1995.

[117] Platform Success Stories. Sal. Oppenheim claims first-mover advantage with a grid solution from IBM and Platform, 2006. Available at http://www.platform.com/solutions/industry/financial-services/success-stories/3597_saloppenheim.pdf.

[118] D. Powell. Distributed Fault Tolerance - Lessons Learnt from Delta-4. In *Papers of the workshop on Hardware and software architectures for fault tolerance : experiences and perspectives*, pages 199–217. Springer-Verlag, 1994.

[119] B. Ramkumar and V. Strumpen. Portable Checkpointing for Heterogenous Architectures. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

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

[121] S. Rao, L. Alvisi, and H. Vin. Hybrid Message Logging Protocols For Fast Recovery. In *Digest of FastAbstracts of the 28th Fault-Tolerant Computing Symposium (FTCS)*, pages 41–42, 1998.

[122] S. Rao, L. Alvisi, and H. M. Vin. Egida : An Extensible Toolkit for Low-Overhead Fault-Tolerance. In *Symposium on Fault-Tolerant Computing*, pages 48–55, 1999.

[123] M. Rieker, J. Ansel, and G. Cooperman. Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux. In *PDPTA*, pages 492–498, 2006.

[124] G. Rodríguez, P. González, M. J. Martín, and J. Tourino. Enhancing Fault-Tolerance of Large-Scale MPI Scientific Applications. In V. E. Malyshkin, editor, *PaCT*, volume 4671 of *Lecture Notes in Computer Science*, pages 153–161. Springer, 2007.

[125] G. Rodriguez, M. J. Martín, P. González, and J. Tourino. Controller/Precompiler for Portable Checkpointing. *IEICE Transactions on Information and Systems, Special Issue on Parallel/Distributed Computing and Networking*, E89-D(2) :408–417, feb 2006.

[126] G. Rodríguez, M. J. Martín, P. González, J. Tourino, and R. Doallo. CPPC : A compiler-assisted tool for portable checkpointing of message-passing applications. *Concurrency and Computation : Practice & Experience*, 22(6) :749–766, apr 2010.

[127] J. F. Ruscio, M. A. Heffner, and S. Varadarajan. DejaVu : Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[128] S. Vialle and X. Warin and C. Makassikis. Large Scale Distribution of Stochastic Control Algorithms for Financial Applications. *The First Workshop on Parallel and Distributed Computing in Finance (Computational Finance) (PDCoF08), IPDPS international conference*, 2008.

[129] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework : System-Initiated Checkpointing. *In LACSI Symposium*, 2003.

[130] N. Santoro. *Design and Analysis of Distributed Algorithms*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2006.

[131] R. D. Schlichting and F. B. Schneider. Fail-Stop Processors : An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3) :222–238, 1983.

[132] F. B. Schneider. Byzantine generals in action : implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2) :145–154, 1984.

[133] F. B. Schneider. The fail-stop processor approach. *Concurrency control and reliability in distributed systems*, pages 370–394, 1987.

[134] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach : A Tutorial. *Computing Surveys*, 22(4) :299–319, 1990.

[135] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. *Lecture Notes in Computer Science (LNCS)*, 2789 :214–223, 2003.

[136] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 249–258. IEEE Computer Society, 2006.

[137] B. Schroeder and G. A. Gibson. Understanding Failures in Petascale Computers. In *Journal of Physics : Conference Series*, volume 78, 2007.

[138] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and Evaluation of a Scalable Application-Level Checkpoint-Recovery Scheme for MPI Programs. In *Proceedings of the 17th ACM/IEEE Supercomputing Conference (SC)*, 2004.

[139] M. Seager. Operational machines : ASCI White, 2003. 7$^{\text{th}}$ Workshop on Distributed Supercomputing (SOS7).

[140] L. M. Silva, J. G. Silva, S. Chapple, and L. Clarke. Fault-Tolerance on Regular Decomposition Grid Applications. In *Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing*, pages 358–365, Washington, DC, USA, 1995. IEEE Computer Society.

[141] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI : The Complete Reference*, volume 1. MIT Press, 1998.

[142] G. Stellner. CoCheck : Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 526–531, 1996.

[143] T. Sterling, D. J. Becker, J. Salmon, and D. F. Savarese. *How to Build a Beowulf : A Guide to the Implementation and Application of PC Clusters*. MIT Press, 1999.

[144] N. Stone, J. Kochmar, R. Reddy, J. R. Scott, J. Sommerfield, and C. Vizino. A Checkpoint and Recovery System for the Pittsburgh Supercomputing Center Terascale Computing System. Technical report, Pittsburgh Supercomputing Center, Pittsburgh, PA 15213, 2001.

[145] N. Stone, J. Scott, J. Kochmar, J. Sommerfield, R. Subramanya, R. Reddy, and K. Vargo. Mass Storage on the Terascale Computing System. In *Proceedings of the 18th IEEE Symposium on Mass Storage Systems and Technologies*, pages 67–, 2001.

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

[147] V. Strumpen. Portable and Fault-Tolerant Software Systems. *IEEE Micro*, 18(5) :22–32, 1998.

[148] Cactus 4.0 Reference Manual. http://cactuscode.org/documentation/ReferenceManual.pdf.

[149] Thorn Documentation : PUGH. http://cactuscode.org/documentation/thorns/CactusPUGH-PUGH.pdf.

[150] I. Thomson. AMD launches 12-core Opteron 6100 series. *The Inquirer*, mar 2010.

[151] Z. Tong, R. Y. Kain, and W. T. Tsai. A Low Overhead Checkpointing and Rollback Recovery Scheme for Distributed Systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, 1989.

[152] Unified CodeCount (UCC). http://sunset.usc.edu/research/CODECOUNT/. Release 2009.10.

[153] S. S. Vadhiyar and J. Dongarra. SRS : A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. *Parallel Processing Letters*, 13(2) :291–312, 2003.

[154] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 62–71. ACM, 2001.

[155] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8) :103–111, 1990.

[156] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Softw. Pract. Exper.*, 28(9) :963–979, 1998.

[157] P. Vezolle, S. Vialle, and X. Warin. Large Scale Experiment and Optimization of a Distributed Stochastic Control Algorithm. Application to Energy Management Problems. In *International workshop on Large-Scale Parallel Processing (LSPP)*, may 2009.

[158] S. Vialle, X. Warin, and P. Mercier. A N-dimensional Stochastic Control Algorithm for Electricity Asset Management on PC cluster and Blue Gene Supercomputer. In *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing PARA*, Trondheim Norvège, may 2008.

[159] D. Watts and R. Moon. IBM System x3850 M2 Technical Introduction, nov 2007. IBM Redpaper, available at www.redbooks.ibm.com/redpapers/pdfs/redp4362.pdf.

[160] E. W. Weisstein. "pi formulas." from mathworld–a wolfram web resource, apr 2011.

[161] Wikipedia. Source lines of code — Wikipedia, The Free Encylcopedia. http://en.wikipedia.org/wiki/Source_lines_of_code, 2010. Online ; accessed 10-Mai-2010.

[162] XtreemOS – Enabling Linux for the Grid. http://www.xtreemos.eu/.

[163] G. Zheng, C. Huang, and L. V. Kalé. Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++. *ACM SIGOPS Operating Systems Review : Operating and Runtime Systems for High-end Computing Systems*, 40(2), apr 2006.

[164] J. F. Ziegler. Terrestrial Cosmic Rays. *IBM Journal of Research and Development*, 40(1) :19–39, 1996.