Réplication de données pour la tolérance aux pannes dans un support d'exécution distribué à base de tâches

Romain Lion

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE

MATHÉMATIQUE ET INFORMATIQUE

Par **Romain LION**

Réplication de données pour la tolérance aux pannes dans un support d'exécution distribué à base de tâches

Sous la direction de : **Samuel THIBAULT**

Soutenue le 13 Décembre 2022

Membres du jury :

Cédric BASTOUL	Professeur	Univ. de Strasbourg	Rapporteur
Leonardo BAUTISTA GOMEZ	Maître de Conférences	BSC	Examinateur
Franck CAPPELLO	Professeur	ANL	Rapporteur
Camille COTI	Professeure	Univ. de Québec	Examinatrice
Luc GIRAUD	Directeur de Recherche	INRIA	Président du jury
Amina GUERMOUCHE	Maîtresse de Conférences	ENSEIRB	Examinatrice
Pierre LEMARINIER	Ingénieur	ATOS SE	Examinateur
Samuel THIBAULT	Professeur	Univ. de Bordeaux	Directeur de thèse

« Restaurer un édifice, ce n'est pas l'entretenir, le réparer ou le refaire, c'est le rétablir dans un état complet qui peut n'avoir jamais existé à un moment donné. »

Eugène Viollet-Leduc

Remerciements

Je tiens à commencer par remercier l'Inria de Bordeaux, qui arrive à proposer une cadre de travail idéal pour mener à bien une thèse. Je pense que l'ambiance générale de l'établissement contribue pour beaucoup à la réussite des doctorants des différentes équipes. Je tiens donc à remercier l'ensemble des équipes créant cet environnement propice à terminer ses études et à entrer en carrière. Je souhaite également remercier l'ensemble de l'open-space de l'équipe Storm, quelle que soit l'année à laquelle j'ai pu y être. Même si les doctorants, ingénieurs et stagiaires vont et viennent, les têtes changent mais l'ambiance reste la même. Merci également aux permanents de l'équipe, qui sont une source de remotivation pour beaucoup de doctorants, toujours à nous encourager pour nous aider à atteindre nos buts. Enfin, merci à notre collègue Bonzini d'être revenu nous assister dans nos travaux après deux années sabatiques.

Je voudrais également dire merci à ma famille pour le soutien qu'ils ont su m'apporter durant toute la thèse mais également les années précédentes. Merci à Sissa et Chew-Chew pour apprécier avec moi le quotidien, et de m'avoir supporté durant ces quatre annèes de thèse. Merci à mes parents pour leur bienveillance et leur valeurs, et merci mon petit cochon pour ton amour vache. Merci aux teutrais pour partager les bons moments derrière les écrans malgré la distance, merci à la P18-APP pour les trop rares mais si appréciables retrouvailles, merci aux MDT pour m'avoir aidé à mettre un terme à la guerre du chant des dragons et merci à tous les autres.

Enfin merci aux membres du jury qui m'auront permis d'avoir un point de vue externe sur mes travaux, ce qui était difficile à obtenir en réalisant une thèse sous fond de confinement et de conférences en visio. Pour finir merci à Samuel pour le temps passé à travailler ensemble, afin de mener à bien les recherches qui ont abouti à ce manuscrit... qui ne s'appelle définitivement pas un tapuscrit.

Réplication de donnée pour la tolérance aux pannes dans un support d'exécution distribué à base de tâches

Résumé : À mesure que la puissance de calcul des nouveaux supercalculateurs augmente, leur fiabilité décroît inexorablement. En effet les limites sont repoussées en augmentant le nombre de composants ainsi que leur complexité, et les systèmes de calcul expérimentent des défaillances au quotidien. La problématique est donc de pouvoir se prémunir des pannes, tout en limitant l'impact sur les performances qu'impose un mécanisme de tolérance aux pannes. Par ailleurs, la complexité de l'architecture des supercalculateurs rend plus difficile leur programmation, ce à quoi tentent de répondre les supports d'exécution à base de tâches comme StarPU. Les travaux présentés ici proposent une solution de tolérance aux pannes adaptée à StarPU. Le modèle de programmation STF utilisé dans StarPU nous permet de créer des sauvegardes asynchrones qui n'interrompent pas le calcul, et qui ne nécessitent aucune synchronisation entre les nœuds de calcul ; ce comportement est atteint en insérant statiquement des requêtes de sauvegarde dans le code d'application, correspondant à une solution de checkpoint de niveau application. Également, gérer des sauvegardes à travers StarPU permet de mettre en commun les données de calcul et les données de sauvegarde, ce qui nous permet de réduire considérablement la quantité de données qui doivent effectivement être sauvegardées. Nous avons exploité cet effet en utilisant les autres nœuds de calcul comme support de sauvegarde, ainsi qu'un mécanisme de message logging afin de redémarrer uniquement le nœud en panne. L'efficacité de cette solution a été évaluée avec une application de décomposition de Cholesky, et nous avons pu voir dans un cas idéal que notre approche permet de tolérer les pannes considérées sans qu'aucune sauvegarde ne soit effectuée en pratique.
Mots-clés : Tolérance aux pannes, Checkpoint, Support d'exécution à base de tâches

Data replication for failure tolerance in a task-based runtime system

Abstract: While computing power of systems grows, their reliabity decreases inevitably. Indeed, performance is achieved by leveraging components quantity and complexity, therefore computing systems are subject to failures on a daily basis. The problem is to use a mechanism to tolerate failures while having the least impact on performance. Moreover, supercomputers archichecture become more and more complex, and so becomes their coding. Data-based runtime systems such as StarPU are responding to this problematic. This thesis proposes a dedicated failure tolerance protocol to StarPU. The STF programming model used in StarPU allows to create consistent coordinated non-blocking asynchronous checkpoints very simply, by inserting checkpoint requests statically in the source code, like an application-based checkpoint solution. Furthermore, managing the checkpoints inside StarPU allows to use the synergy between computing data and checkpoint data, allowing to significantly reduce the amount of data that needs to be saved. We exploit this effect by choosing to save checkpoints on the other computing nodes, and by performing local rollback using message logging. The efficiency of our proposal is evaluated with a Cholesky decomposition application. We also show that with a particular setting for this application, our approach allows to tolerate the failure corresponding to our hypothesis without having any data actually replicated on other nodes. This is done by using the fact that the application already replicates enough data due to the computation needs, while with our approach we are able to exploit these data as checkpoint data.
Keywords: Failure tolerance, Checkpoints, Task-based runtime system

Table des matières

Table des figures

Glossaire

Checkpoint global Collection de checkpoints locaux composée d'un checkpoint local par nœud.

Checkpoint global cohérent Checkpoint global libre de tout message orphelin. Dans nos travaux un checkpoint global n est constitué des checkpoints locaux n de chacun des nœuds.

Checkpoint local Checkpoint réalisé de manière locale par un nœud sur ses nœuds backup, et utilisable pour redémarrer un seul nœud.

Message en-transit Message qui est à la fois considéré comme envoyé et non-reçu par différentes partie d'un système. Si le message n'a pas lieu il sera perdu.

Message orphelin Message qui est à la fois considéré comme reçu et non-envoyé par différentes parties d'un système. Cela crée une incohérence.

Nœud backup Nœud qui contribue au checkpoint d'un nœud d'origine. Le nœud d'origine peut avoir plusieurs nœuds backup s'il choisit de séparer son checkpoint sur différents supports.

Nœud backup principal Nœud backup qui a un rôle particulier dans la sauvegarde. En soumettant plusieurs templates lors d'un checkpoint, on aura autant de nœuds backup principaux que de templates.

Nœud d'origine Nœud qui a émis un checkpoint vers son ou ses nœuds backup.

Template / Checkpoint template Structure permettant de définir quelles données doivent être sauvegardées, et où elles doivent être sauvegardées. On utilisera autant de templates que le nombre de checkpoints répliqués souhaités.

Introduction

La fiabilité des super-calculateurs décroît quand leur puissance augmente. Cela s'explique par le simple fait que les performances sont atteintes davantage en augmentant le nombre de ressources de calcul qu'en augmentant les performances d'une ressource. Les machines sont donc de plus en plus complexes, et si le taux de pannes pouvait être encore négligé il y a une décennie, ce n'est plus le cas aujourd'hui. Néanmoins se protéger contre les pannes nécessite d'utiliser des ressources utilisées d'ordinaire par le calcul, et ne peut que le ralentir. Utiliser un mécanisme de tolérance aux pannes dégrade ainsi inexorablement les performances. Afin de pouvoir être acceptable pour un utilisateur, la tolérance aux pannes ne doit pas dégrader les performances outre mesure ; on évitera donc de l'utiliser si cela impose par exemple au programme d'être deux fois moins rapide, alors que l'application a neuf chances sur dix de terminer sans subir de panne. On observe donc qu'il y a une relation entre le prix à payer pour ne pas interrompre un calcul en cas de panne, et les probabilités de subir une panne. Comme la probabilité de subir une panne est plus forte sur des nouveaux supercalculateurs, l'efficacité d'une solution pour se prémunir contre des pannes doit être encore plus grande que ce que l'on acceptait par le passé. En effet le surcoût induit par la tolérance aux pannes est directement proportionnel au nombre de machines et la quantité de données qu'elles manipulent, et ce sont précisément des grandeurs qui ne cessent de croître à chaque nouvelle machine de calcul. On a donc des machines qui deviennent de plus en plus sujettes aux pannes, alors qu'il est de plus en plus difficile de se prémunir de ces pannes.

StarPU est un support d'exécution à base de tâches, et existe dans une version distribuée appelée StarPU-MPI. Aucun travail à propos de la tolérance aux pannes n'avait encore été réalisé pour le cas de StarPU avant cette thèse. Les travaux présentés ici ont été réalisés dans le cadre du projet Européen Exa2Pro[55], qui avait pour but de proposer une pile logicielle adaptée à la programmation des machines Exascale. C'est donc naturellement que les thématiques de tolérances aux pannes ont été abordées et que les travaux de cette thèse ont été initiés. StarPU est un support d'exécution qui propose d'écrire une application en soumettant des tâches de manière séquentielle. Ces tâches peuvent avoir plusieurs implémentations afin d'être exécutées sur ressources de

calcul hétérogène. StarPU va s'occuper de lancer les tâches de manière asynchrone, en respectant les dépendances induites par l'aspect séquentiel de la soumission, tout en parallélisant les tâches qui peuvent l'être. La version distribuée StarPU-MPI nécessite une distribution des données initiales sur l'ensemble des machines de la part de l'utilisateur. Le reste du code est identique, on garde une soumission séquentielle des tâches, et StarPU-MPI va lancer les tâches de la même manière, tout en automatisant les communications déduites des dépendances des tâches. On peut donc écrire facilement des codes parallèles avec cette approche, en permettant une description du calcul qui est quant à elle séquentielle. Mais le fait que StarPU automatise autant de choses rend l'état global non-déterministe ; d'une exécution à l'autre, l'ordre d'exécution des tâches parallèles est différent, l'ordre d'envoi diffère lui aussi... Sachant cela on peut se demander comment l'on peut réussir lors d'une panne à reprendre un calcul pour une application qui se comporte ainsi.

Beaucoup de travaux ont été effectués pour rendre ce genre d'applications tolérantes aux pannes, mais les résultats à attendre ne seront pas bons. Tous reposent sur la prise de sauvegardes régulières, appelées checkpoints, mais cela nécessite a priori d'interrompre simultanément l'ensemble des machines afin de garantir que le checkpoint représente un état global cohérent. D'autres approches permettent de produire des checkpoints sans synchronisation mais reposent sur des algorithmes difficiles à mettre en pratique. De plus ces approches nécessitent de sauvegarder toute l'empreinte mémoire de l'application, des bibliothèques utilisées... on aimerait donc utiliser une autre solution si cela est possible. En effet il existe des approches où l'on explicite les sauvegardes dans le code d'application. Mais en plus de devoir modifier le code de l'application, cela interrompt la soumission de tâches à chaque checkpoint, ce qui impose d'attendre la fin de toutes les tâches soumises avant le checkpoint ou de nouvelles tâches. Le principe de cette approche dégrade les performances sans compter le coût des sauvegardes, d'autant plus dans StarPU car l'on retarde explicitement des tâches qui pourraient être lancées plus tôt. Nous avons donc cherché une approche qui puisse ne pas demander de compromis dès le départ. On a également pu remarquer que dans un calcul distribué, des données sont transférées entre les nœuds et se retrouvent déjà naturellement répliquées par le calcul. Il serait très intéressant de se servir de ces données afin de reprendre un calcul si un nœud tombe en panne, et donc il serait dommage de perdre du temps à les sauver ailleurs alors qu'elles sont déjà répliquées.

Les travaux présentés ici proposent une solution adaptée au support d'exécution StarPU. En utilisant ses propriétés et les optimisations déjà disponibles, il est possible de définir des checkpoints vraiment efficacement. Grâce à la sémantique *Sequential Task Flow* (STF), on verra qu'il est possible de couper le graphe des tâches exécutées, afin de définir des points de reprise. Cela nous permet de garantir des checkpoints globaux

cohérents, sans besoin de coordination entre les nœuds. De plus en utilisant le suivi des modifications de données de StarPU, il est possible de sauver de manière différentielle les checkpoints. Comme l'asynchronisme est omniprésent dans StarPU, on pourra également faire en sorte que chaque checkpoint soit réalisé de manière incrémentale, c'est-à-dire étalée dans le temps. Mais surtout on pourra réaliser cela sans empêcher le calcul de progresser avant et pendant un checkpoint. Nous verrons également comment nous pouvons réutiliser en tant que données de sauvegardes les données déjà répliquées par le calcul, dans un cadre où l'on souhaite redémarrer uniquement le calcul du nœud qui est tombé en panne, en se basant sur une technique de message-logging. Toutes ces propriétés nous permettront, dans une application de décomposition de Cholesky, en utilisant des propriétés qui y sont spécifiques, de pouvoir aller jusqu'à faire des sauvegardes qui ne coûtent que quelques kilo-octets, plutôt que des dizaines de giga-octets.

Après avoir établi le contexte et fait un bilan sur l'état de l'art du domaine de la tolérance aux pannes, nous proposerons une description de StarPU, de son fonctionnement en exposant les propriétés que nous pourrons réutiliser par la suite. Nous pourrons ensuite présenter comment il est possible de combiner les différentes approches de l'état de l'art en les adaptant aux propriétés de StarPU, afin de pouvoir proposer une solution de checkpoints efficace. Nous présenterons les propriétés de ces checkpoints, ainsi que des mesures permettant de valider notre approche. Ensuite nous présenterons comment il est possible de redémarrer uniquement le nœud en panne, autrement appelé redémarrage local, en utilisant une technique de message logging, qui elle aussi peut profiter des propriétés de StarPU pour limiter l'impact sur les performances et l'encombrement mémoire. Enfin nous conclurons ces travaux en apportant des perspectives de recherches liées à la solution proposée.

Chapitre 1

Contexte et État de l'art

De nos jours le calcul scientifique est un outil indispensable utilisé dans de nombreux domaines, allant de la simulation météorologique globale jusqu'à l'optimisation de la consommation d'une montre connectée. Si nous sommes aujourd'hui capables d'effectuer de tels calculs reproduisant des phénomènes physiques avec la fidélité nécessaire, c'est notamment grâce aux avancées technologiques et théoriques du siècle dernier dans quatre piliers fondateurs. Typiquement, le physicien crée (1) un modèle reproduisant un phénomène réel en utilisant des outils mathématiques, puis pour évaluer ce modèle il faut utiliser (2) une méthode numérique adaptée, d'où va découler (3) un algorithme qui est ensuite (4) exécuté sur des machines de calculs. Cette méthode de travail met en évidence que la grande quantité de compétences nécessaires à l'élaboration d'un programme de calcul requiert plusieurs corps de métier pour atteindre un calcul rapide et mathématiquement juste.

Durant les dernières décennies des solutions ont été développées afin de faciliter les interactions entre les différentes couches ; c'est le cas notamment pour l'interaction entre l'algorithme et la machine de calcul. Des supports d'exécution sont mis à disposition des scientifiques afin de faciliter le développement de leurs algorithmes ; ceux-ci leur permettent de réduire la complexité du travail et d'améliorer le rapport entre temps de développement et performance du programme. StarPU [6] est un support d'exécution qui partage cet objectif. C'est un outil libre développé par l'équipe Storm de l'Inria Bordeaux, qui est le cadre de développement de nos travaux, dont nous parlerons plus en détail dans la section 1.3.

Même à supposer qu'à l'issue de la phase de conception du programme toute erreur soit évacuée par débuggage et que l'on ait donc un programme parfaitement fonctionnel, il n'est pas dit pour autant que ce dernier fournisse le résultat attendu lors de son exécution. En effet, même s'il peut paraître pertinent de considérer qu'une machine de calcul fonctionne comme attendu en tout instant, il existe toujours une probabilité qu'une erreur se produise ou qu'un de ses composants tombe en panne. Bien que la

probabilité d'une interruption de l'exécution par une erreur fortuite paraisse tout à fait négligeable sur une machine de calcul tel qu'un simple ordinateur personnel, le phénomène est tout autre lorsque l'on distribue le calcul sur des dizaines de milliers de machines. En effet, dans un souci d'améliorer le temps de calcul, on cherche à paralléliser le travail sur autant de machines que faire se peut, chaque machine effectuant une sous-partie du calcul. Mais alors si l'on s'attarde sur la fiabilité globale d'un système où chaque machine est essentielle, on peut se rendre compte avec l'équation 1.1 que l'ordre de grandeur du temps moyen de fonctionnement avant interruption ($MTTI$: Mean Time To Interruption) de l'ensemble du système n'est plus du tout négligeable.

$$MTTI_{sys} = (\sum_{i=0}^{n-1} \frac{1}{MTTI_i})^{-1} \tag{1.1}$$

En considérant que chaque élément a une fiabilité identique, on peut simplifier la formule en la forme de l'équation 1.2.

$$MTTI_{sys} = \frac{MTTI_i}{n} \tag{1.2}$$

On peut alors voir que la fiabilité globale d'un système à n machines est fortement réduite lorsque n est grand. Typiquement le $MTTI$ d'une machine est de l'ordre de 10^4 h [22] et le $MTTI$ du système est alors de l'ordre de l'heure lorsque l'on travaille avec 10^4 machines et davantage (le super-calculateur Frontier qui est le plus puissant en date de Juin 2022 est composé de plus de 9 400 nœuds de calculs [57]). La fiabilité d'une seule machine peut difficilement être améliorée, il est ainsi certain que le taux d'interruptions ne fait que croître avec l'augmentation de la puissance de calcul de nos super-calculateurs.

Le terme de *résilience* englobe le domaine des solutions qui permettent de traiter ces problèmes. Il s'agit sémantiquement de la capacité à surmonter une interruption fortuite. Quand on est en présence d'une interruption que l'on est capable de surmonter on ne parle plus de $MTTI$ mais de $MTBI$ (Mean Time Between interruption - Temps moyen entre interruptions), étant donné que l'on traite l'interruption pour que le système continue de fonctionner jusqu'à la prochaine interruption. Dans le cadre de la tolérance aux *pannes*, on parle plus spécifiquement de $MTBF$ (Mean Time Between Failures - Temps moyen entre pannes).

La grande majorité des solutions de tolérance aux pannes que l'on trouve dans l'état de l'art utilisent un modèle de panne très large, qui prend en compte le fait que n'importe quel scénario de panne peut intervenir, sans prendre en compte la probabilité d'occurrence de ces scénarios. Agir ainsi permet de se prémunir contre tout type de pannes, apportant un taux d'échec théoriquement nul, et un $MTTUF$ théoriquement

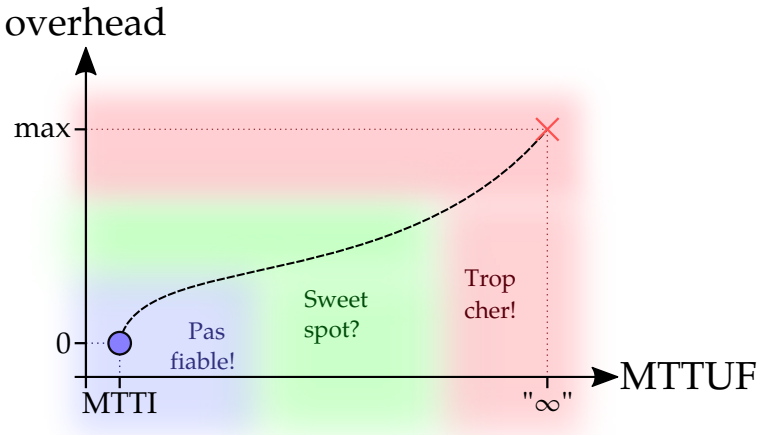


FIGURE 1.1 : Illustration du compromis entre le surcoût d'une solution de tolérance aux pannes et l'apport sur le temps moyen jusqu'à une panne fatale.

infini (Mean Time To Unrecoverable Failure - Temps moyen avant une panne fatale). Néanmoins utiliser ces propositions nécessitent de faire des sauvegardes ce qui impacte les performances de calcul de manière non-négligeable, et qui peut décourager de nombreux utilisateurs de faire l'effort d'adopter une solution de tolérance aux pannes. De plus on observe de manière générale que les solutions de tolérance existantes ont un coût proportionnel au nombre d'unités de calcul, qui on le rappelle est un des paramètres principaux qui permet d'augmenter la puissance de calcul des super-calculateurs. On peut se demander s'il est intéressant de considérer uniquement les pannes les plus probables à condition qu'il soit possible de les traiter à l'aide d'une solution considérablement moins coûteuse. Dans ce cas on abandonne le fait que l'on est immunisé à tout scénario de pannes, ce qui réduit le $MTTUF$ et qui n'est donc plus théoriquement infini. Mais si le $MTTUF$ reste très largement supérieur à la durée d'exécution du programme que l'on cherche à protéger des pannes, la faible probabilité de panne ajoutée est à relativiser face à la diminution du surcoût apporté par la tolérance aux pannes. On parle d'*overhead* lorsque l'on évoque le surcoût apporté par une fonctionnalité.

Pour illustrer ce propos on peut représenter sur la Figure 1.1 le surcoût inexistant si l'on n'utilise aucune solution de tolérance aux pannes (cercle bleu). Étant donné que l'on ne tolère aucune panne, le temps moyen jusqu'à une panne fatale sera le $MTTI$ du système de calcul. Le surcoût d'une solution de tolérance aux pannes avec un $MTTUF$ en théorie infini est représenté par la croix rouge. On peut arbitrairement interpoler une courbe entre ces deux points en faisant la supposition qu'elle a une pente monotone croissante quelconque. Dans ce cas il existe certainement un meilleur compromis à utiliser, permettant d'avoir un $MTTUF$ suffisamment grand pour être conforme aux besoins, tout en ayant un overhead plus appréciable que les solutions usuelles. Néanmoins opter pour ignorer certaines pannes nécessite d'avoir une bonne connaissance quant à la classification de panne et à leur taux d'occurrence, ce qui tend à manquer dans les

études. Mais le domaine commence tout de même à s'intéresser à ce principe, et l'on voit apparaître des solutions de tolérances au pannes sur plusieurs niveaux, permettant de traiter de manière efficace les pannes selon leur taux d'occurrence respectif. C'est la proposition de FTI par exemple, dont on parle en section 1.2.2.3.

Pour présenter plus en détail le contexte, commençons par évaluer quelles sont les spécificités du domaine du calcul distribué. En ayant cela à l'esprit nous pourrons faire un tour d'horizon des différentes techniques de tolérance aux pannes proposées dans l'état de l'art. Enfin nous nous intéresserons aux spécificités qui définissent un support d'exécution tel que StarPU, et présenterons les propriétés que nous pourrons exploiter dans nos contributions.

1.1 Le calcul distribué

Les besoins en calcul scientifique croissent avec les années, demandant plus de ressources de calcul afin d'améliorer la finesse des résultats. La loi de Moore est de plus en plus difficile à suivre au niveau la taille des transistors [51], on augmente la quantité de cœurs de calcul par processeur et le nombre de processeurs dans le système pour subvenir aux besoins de l'industrie. Par conséquent la course à la puissance est réalisée en augmentant l'échelle des infrastructures de calcul, les systèmes étant ainsi dotés de plus en plus d'unités de calcul. Mais pour bénéficier de la puissance de ces machines il faut parvenir à paralléliser le calcul, c'est-à-dire que chaque machine de calcul va calculer une contribution afin de faire progresser l'exécution globale.

Ainsi des efforts sont réalisés pour trouver des méthodes de calcul largement parallélisables en ne laissant qu'une faible part de calcul séquentiel (la part qui ne peut bénéficier de ce parallélisme). Du point de vue de l'infrastructure il faut considérer un super-calculateur comme un réseau d'ordinateurs appelés *nœuds de calcul*, chacun contribuant à l'avancement du programme. Le calcul est donc distribué sur des nœuds qui doivent tout de même interagir afin de contribuer au résultat. La majorité des programmes HPC utilisent MPI (Message Passing Interface) [65], une interface qui permet aux nœuds d'échanger entre eux des données par messages. Cette interface propose un paradigme SPMD (Single Program Multiple Data) qui permet d'avoir un programme unique pour tous les nœuds, et seules les données vont différer d'un nœud à l'autre.

Une information capitale dans un environnement MPI est le *rang*, identifiant unique à chaque nœud qui permet au programme de savoir quel est son rôle dans un *communicateur MPI*. Le communicateur est une structure MPI composée de n nœuds MPI, rangés de 0 à $n-1$, au sein duquel ils peuvent communiquer. Les développeurs ont alors la charge de définir dans le programme quel est le rôle de chaque nœud, quelles sont les

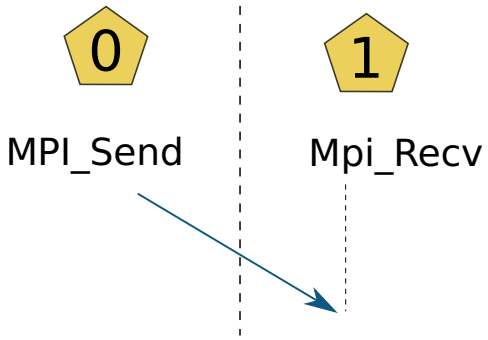


FIGURE 1.2 : Chronogramme de l'exécution du code 1.1 : seul `MPI_Recv` attend la fin de la réception du message.

contributions réalisées par chacun et quels messages doivent être échangés entre quels nœuds. Le code 1.1 est un exemple permettant de voir comment s'écrit et fonctionne un programme MPI. À noter que le nombre de nœuds MPI sur lesquels le programme est distribué est déterminé au lancement de l'exécution avec la commande `mpi_exec`. Cette commande est exécutée une seule fois pour distribuer le programme sur autant de nœuds que voulus. Ici on distribue le programme sur 2 nœuds MPI avec la commande `mpi_exec -N 2 ./a.out`.

Cet exemple de code permet de mettre en évidence l'utilisation de MPI, dont l'exécution est illustrée sur la Figure 1.2. Conformément à l'approche SPMD, le comportement de tous les nœuds est décrit dans le code, mais chaque nœud détermine la portion qu'il doit effectuer grâce au *rang* affecté par MPI. On remarque qu'il n'y a qu'un message envoyé en *point à point* : le nœud 0 envoie le message avec `MPI_Send`, bloquant le programme sur ce nœud jusqu'à ce que l'envoi soit effectué et qu'il soit de nouveau possible d'écrire dans la variable `x`, sans savoir si le nœud récepteur de message a terminé sa réception. Pour s'assurer de l'acquittement du message il faudrait utiliser des appels spécifiques intrinsèquement plus lents. Le nœud 1 reçoit le message avec l'appel `MPI_Recv`, qui va bloquer jusqu'à ce que la variable `x` contienne la valeur du message. Une alternative est d'appeler `MPI_Irecv` qui est immédiat, ce qui permet d'effectuer autre chose en attendant la réception du message ; en contrepartie on ne peut plus travailler avec la variable `x` avant de tester que la réception est terminée avec la fonction `MPI_Test` ou de l'attendre avec l'appel bloquant `MPI_Wait`. Un autre point important est le paramètre `tag` présent dans les appels `MPI_Send` et `MPI_Recv`, ici 0 dans l'exemple. Ce dernier peut être vu comme un "canal de communication" de type FIFO. En pratique `MPI_Recv` ne peut recevoir le message que si le tag correspond à celui d'un des messages entrants. De plus la propriété FIFO apporte une propriété séquentielle dans les messages au sein d'un même "canal" : si le nœud 0 avait envoyé 2 messages au nœud 1 avec le même tag, MPI assure que l'ordre de livraison des messages aux appels de réception est le même que l'ordre d'envoi des messages. En revanche ce n'est plus le cas si les

```c
#include <mpi.h>

int main(int argc, char* argv[]) {
    // Les 2 nœuds exécutent le même programme (SPMD)

    // Initialisation de MPI
    MPI_Init(argc, argv);

    // MPI_COMM_WORLD est le communicator initial contenant les 2 nœuds.

    int rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // rank = 0 pour le 1^er nœud, 1 pour le 2^nd

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // world_size vaut 2 pour les 2 nœuds

    int x;
    if (rank == 0)
    {
        // Exécuté uniquement par le nœud 0
        x = -1;
        MPI_Send(&x, 1 /*nb*/, MPI_INT /*type*/, 1 /*destinataire*/, 0 /*tag*/,
                 MPI_COMM_WORLD);
    }
    else if (rank == 1)
    {
        // Exécuté uniquement par le nœud 1
        MPI_Recv(&x, 1 /*nb*/, MPI_INT /*type*/, 0 /*émetteur*/, 0 /*tag*/,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Node 1 received \%d from node 0\n", x);
    }

    // Point de synchronisation
    MPI_Barrier(MPI_COMM_WORLD);

    // Terminaison de MPI
    MPI_Finalize();
}
```

Code 1.1 : Exemple de code MPI : le nœud 0 envoie un message au nœud 1 qui le réceptionne.

messages sont échangés entre deux même nœuds si les tags ne sont pas identiques : l'ordre de réception ne correspond plus forcément à l'ordre d'envoi.

Enfin MPI propose des *collectives*, qui sont des fonctions de communication MPI mettant à contribution l'ensemble des nœuds d'un communicateur. Dans l'exemple ci-dessus la fonction `MPI_Barrier` est la seule collective, qui permet d'attendre que tous les nœuds du communicateur aient atteint cet endroit du code, ici utilisée pour des besoins de synchronisation et s'assurer ainsi que tous les nœuds ont fini de travailler avant de terminer le programme. D'autres collectives existent permettant notamment d'échanger des données entre les nœuds du communicateur comme par exemple `MPI_Reduce` ou `MPI_Bcast`.

Il faut retenir de cette partie que la grande majorité des programmes de calcul haute performance distribués sont développés à l'aide de MPI. Cet exemple simplifié ne reflète pas la complexité d'une réelle application utilisant des dizaines de milliers de nœuds de calcul. On peut tout de même évoquer le fait que pour parvenir à des performances intéressantes, les programmes MPI peuvent faire preuve d'asynchronisme ; en effet pour pouvoir effectuer une partie du calcul pendant la réception de données nécessaires plus tard, les nœuds peuvent poster plusieurs réceptions non-bloquantes avec `MPI_Irecv` entrelacées pour mieux recouvrir le temps de communication avec du calcul. Cette méthode permet d'atteindre un fort taux de parallélisme entre les communications et le calcul mais devient problématique lorsque l'on souhaite mettre en œuvre une tolérance aux pannes.

1.2 Tolérance aux pannes

Même si la problématique des pannes est de plus en plus préoccupante à cause des caractéristiques des super-calculateurs actuels et futurs, la recherche dans le domaine de la tolérance aux pannes est active depuis près d'un demi-siècle. Il existe en conséquence diverses solutions développées afin de répondre à la problématique des pannes [35, 42, 23, 22, 34, 27, 29, 46]. Malgré une grande variété d'approches, toutes ont le point commun de dégrader les performances en comparaison d'une exécution sans solution de tolérance aux pannes. En effet il s'agit de rajouter de la complexité dans un programme afin de répondre à une spécification qui n'est pas originellement présente. Mais il ne faut pas oublier que pour une exécution à très large échelle, il reste plus intéressant d'exécuter avec succès un programme tolérant aux pannes qui est deux fois plus lent, plutôt que d'échouer systématiquement à exécuter la version plus rapide sans tolérance aux pannes. Afin de faire un tour d'horizon sur le sujet, intéressons-nous d'abord aux généralités permettant de distinguer clairement les caractéristiques du domaine et les enjeux, avant de passer aux principes des solutions existantes.

1.2.1 Généralités

Nous allons voir ici comment le sujet est abordé dans la littérature, en adoptant un point de vue suffisamment large pour couvrir avec pertinence les différents aspects de la problématique. Il s'agit de présenter les principes élémentaires des solutions, de visualiser comment modéliser simplement un système de calcul distribué, en s'intéressant d'abord aux hypothèses de départ.

1.2.1.1 Faute, erreur ou panne ?

Il existe une relation hiérarchique entre les trois phénomènes de faute, d'erreur et de panne.

Une *faute matérielle* se produit lorsqu'un aléa corrompt une donnée. Cette corruption va remettre en cause la justesse d'un calcul, le contenu d'un message ou la cohérence de l'état d'un programme par exemple. La cause de cette corruption est considérée comme fortuite car causée par des perturbations physiques non maîtrisables et imprévisibles. Dans des machines binaires telles que les nôtres, une faute se caractérise en réalité par une inversion d'un ou plusieurs bits, qui va ensuite causer des problèmes à une échelle plus macroscopique. Du moment qu'une faute a eu lieu, il faut considérer le système en *erreur*. Néanmoins cette considération ne peut avoir lieu que si l'on est en capacité de détecter l'erreur ; ainsi l'on peut se trouver dans le cas d'une erreur silencieuse qui va donc perdurer sans symptôme au risque de se propager, ou de provoquer une panne. Si l'erreur est détectée, il est possible de la corriger quand on en est capable, ou d'interrompre le programme dans le cas contraire. Enfin une *panne* peut être due à une erreur que la machine ne peut corriger d'elle-même, ou bien due à une panne matérielle affectant un composant. Dans tous les cas la panne remet entièrement en cause la capacité de la machine à réaliser la tâche qui lui est affectée. Ainsi, *faute* désigne le changement de comportement en lui-même, *erreur* désigne ses conséquences, potentiellement asymptomatiques pendant un certain temps, mais qui peut aboutir, si elle n'est pas traitée, à l'arrêt complet du calcul : la *panne*.

Pour réagir à ces phénomènes il est nécessaire de les détecter. Si une panne est facilement repérable grâce à son ampleur, les fautes sont en revanche plus difficiles à détecter. Même si de nombreux efforts sont réalisés pour détecter au plus tôt ces fautes (Checksum, Error correction code, parity check, coherency check...) certaines passent inexorablement entre les mailles du filet, devenant des erreurs silencieuses se propageant potentiellement à l'ensemble des nœuds et compromettant la justesse du calcul. Le support d'exécution servant de cadre de travail ici (StarPU) ne propose actuellement pas de solutions de détection de ces erreurs silencieuses, mais permet néanmoins à l'application de redémarrer une tâche si le développeur implémente un moyen de détecter la présence de ce type d'erreur par contrôle de cohérence du calcul,

en vérifiant par exemple des valeurs invariantes du calcul comme la quantité totale d'énergie. Pour ces travaux nous faisons la forte hypothèse d'ignorer la présence de ces erreurs silencieuses, en considérant que l'application et les systèmes de détection matériels permettent de réduire l'occurrence de ces erreurs à un taux acceptable.

Dans cette thèse nous nous intéressons aux pannes franches, un arrêt de l'exécution émergeant d'une erreur que le programme n'a pas su gérer, ou provoquée par une panne matérielle d'un des composants. Ce genre de panne se manifeste à une échelle où elle est facilement détectable, et s'y intéresser permet de traiter n'importe quelle faute qui n'a pas pu être gérée en interne et qui a donc été intentionnellement transformée en une panne. En revanche procéder ainsi suppose qu'il n'y a pas de propagation d'erreur de calcul entre les nœuds avant la détection de la faute.

1.2.1.2 Corrélation entre les pannes

Une des hypothèses que l'on retrouve dans une partie des travaux sur le sujet est que l'on considère que les pannes ne sont pas corrélées ; les pannes sont donc supposées indépendantes, distribuées uniformément dans le temps et dans le système. Aupy et al.[8] montrent que bien que ce point de vue ne soit pas totalement représentatif des scénarios de pannes réels, supposer l'indépendance des pannes est la meilleure manière d'aborder le problème. Les analyses de cet article montrent en effet que les algorithmes capables de gérer au mieux la particularité des pannes en cascade proposent des gains au mieux dérisoires en comparaison de l'existant. C'est pour cela que dans cette thèse, nous considérons également que les pannes sont indépendantes.

1.2.1.3 Modélisation série ou parallèle d'un système

Nous évoquions en début de ce chapitre des équations permettant de définir le temps moyen avant interruption d'un système de calcul. Les équations 1.1 et 1.2 reposent en réalité sur une hypothèse ; on cherche à exploiter une machine au mieux, à savoir que chaque élément est d'une importance critique. Chaque nœud de calcul va effectuer une partie du calcul et échanger les résultats intermédiaires avec les autres nœuds afin de contribuer à la progression. En procédant ainsi, on s'expose à une vulnérabilité : chaque nœud de calcul effectue un travail qui lui est propre et la panne d'un de ces nœuds est bloquante pour la progression. On a donc une modélisation "série", où chaque élément du système est important. On se rend alors compte que cette vulnérabilité vient de l'importance accordée à chaque nœud, et que réduire cette importance réduirait également cette vulnérabilité. Il est possible de modéliser le système différemment en adoptant un point de vue "parallèle", ou plus particulièrement une combinaison série/parallèle ; cela revient à avoir des composants parallèles prenant le relais en cas de panne, ce qui n'est plus compatible avec les équations 1.1 et 1.2. Une façon évidente de

réaliser cela est de répliquer le calcul sur plusieurs nœuds.

1.2.1.4 Réplication du calcul

Afin de réduire la vulnérabilité précédemment évoquée, on peut dupliquer le travail d'un nœud sur plusieurs machines [39]. Redonder ainsi le calcul sur un nombre de k machines signifie également diviser la puissance de calcul par ce même nombre k et ainsi augmenter considérablement le temps d'exécution d'une application. Typiquement on duplique le travail d'un nœud sur 2 ou 3 machines, ce qui permet de fiabiliser l'exécution. L'intérêt est de pouvoir compter sur une autre machine dans le cas où l'une d'entre elles tombe en panne. Dans le cas d'une réplication sur k machines, il faudrait que les k machines effectuant le même travail tombent toutes en panne pour être dans l'impossibilité de continuer l'exécution. Cette méthode permet également d'effectuer une comparaison entre les résultats intermédiaires des différents réplicas, et l'on a alors un moyen efficace de détecter des erreurs silencieuses, voire de les corriger si $k \geq 3$.

$$MTTI_{sys\ replic.} = \frac{1 + 4^b \frac{(b!)^2}{(2b)!}}{2b \times MTTI_i} \quad \text{avec } 2b = n \tag{1.3}$$

En considérant les pannes comme étant non corrélées et donc des événements indépendants, on peut voir d'après des travaux de Benoît et al. [11] que la fiabilité d'un système de calcul augmente considérablement en effectuant une duplication ($k = 2$), travaux dont est issue l'équation 1.3. En début de ce chapitre nous obtenions un $MTTI_{sys} = 1$ h sur un système avec $n = 10^4$ machines ayant un $MTTI_i = 10^4$ h ; si l'on effectue une duplication ($k = 2$) et que l'on travaille avec $b = 5.10^3$ paires de nœuds, on obtient un $MTTI_{sys\ replic.}$ de 178 h d'après l'équation 1.3, bien que l'on doive en contrepartie sacrifier la moitié de la puissance du système. On remarque notamment que le gain relatif en fiabilité $G = \frac{MTTI_{sys\ replic.}}{MTTI_{sys}}$ est d'autant plus grand qu'il y a de machines dans le système ($G = 3$ pour $n = 2$ machines, $G = 178$ pour $n = 10^4$, $G = 561$ pour $n = 10^5$). Mais le compromis de la performance est difficile à accepter à moins de concevoir les machines en conséquence ; Engelmann et al. [36] proposent d'intégrer davantage d'éléments de calcul en sacrifiant leur fiabilité au profit de la quantité. En effet en partant du principe qu'un processeur coûte bien moins cher si l'on diminue son MTTI de 10^5 h à 10^2 h, on pourrait augmenter considérablement le nombre de nœuds de calcul pour un budget relativement identique. Il serait ainsi possible d'effectuer une redondance du calcul plus efficace par rapport à un système composé de moins de machines, même si ces dernières sont paradoxalement plus fiables.

Malheureusement le sacrifice de la fiabilité des processeurs ne permet pas de réduire suffisamment les coûts en pratique. Ainsi même si les avantages dont on bénéficie en contrepartie permettent de répondre efficacement à la problématique des pannes, la

réplication de calcul paraît être trop coûteuse, notamment du point de vue énergétique. De plus pour être réalisables en pratique, les travaux cités ci-dessus combinent la réplication de calcul avec d'autres méthodes de tolérance aux pannes, notamment les *checkpoints* dont nous parlons ci-dessous.

1.2.1.5 Sauvegarder ou recalculer

Si l'on occulte la solution de la réplication de calcul, on a alors une modélisation série du système comme évoqué en Section 1.2.1.2. Dans ce cas chaque nœud de calcul effectue une contribution unique au calcul. Chacun des nœuds possède une forte importance dans le réseau : si un nœud tombe en panne, l'ensemble du système de calcul est compromis. Certains programmes peuvent néanmoins continuer l'exécution si une contribution est perdue, c'est le principe de l'ABFT (Algorithm Based Fault Tolerance) [32].

Mais pour la vaste majorité des applications, une contribution manquante provoque l'impossibilité de compléter l'exécution. On doit donc garantir la disponibilité de la contribution en cas de panne de son auteur, et pour cela deux solutions s'offrent à nous :

— Recalculer la contribution qui a été perdue, ce qui est coûteux en temps. On peut recommencer du départ le calcul d'un seul nœud, mais celui-ci nécessitera très certainement des contributions d'autres nœuds, qui ne sont en général plus disponibles ;

— Récupérer une sauvegarde de la contribution, ce qui est plus compliqué qu'il n'y paraît et coûteux en espace mémoire. Pour s'assurer qu'aucune contribution ne manque il faudrait sauvegarder toutes les contributions ce qui n'est en pratique pas réalisable à de larges échelles.

Chacune de ces deux solutions est en réalité impossible à mettre en pratique seule car trop coûteuse, et l'on va préférer une combinaison des deux, i.e. effectuer des sauvegardes de temps en temps pour permettre de recalculer des contributions perdues à partir d'un état plus avancé que l'état initial. C'est ce que l'on appelle les *checkpoints*.

1.2.2 Checkpoints

Comme illustré sur la Figure 1.3, un checkpoint (CP) est simplement une sauvegarde de données qui permet de reprendre l'exécution à partir de l'état sauvegardé, qui est donc un état avancé du calcul. En cas de panne, il n'est ainsi nécessaire de ré-exécuter que les calculs ayant été effectués entre la dernière sauvegarde et la panne. Il s'agit d'apporter un compromis entre la mémorisation des résultats intermédiaires du calcul et le re-calcul de ces résultats. Mais pour un programme informatique, redémarrer dans un état avancé n'est pas si évident. Intéressons-nous d'abord au cas d'un programme isolé avant d'élargir la problématique aux programmes distribués sur plusieurs machines.

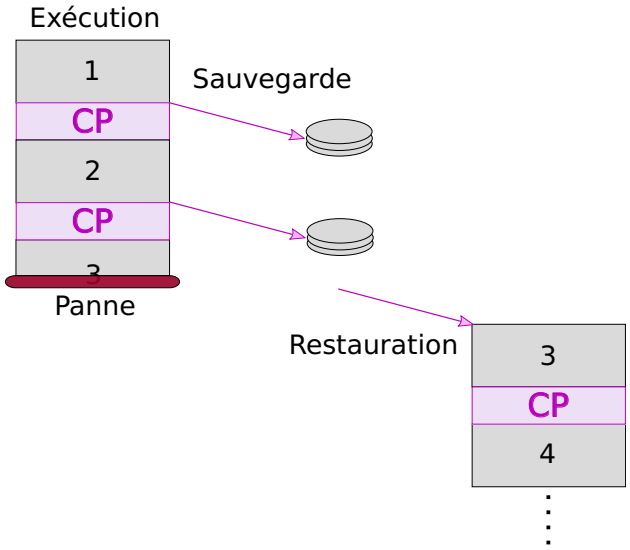


FIGURE 1.3 : Principe de checkpoint : après chaque étape de calcul (en gris) un checkpoint est réalisé (en mauve), ce qui permet en cas de panne (ici durant la troisième étape de calcul) de reprendre le calcul depuis l'état sauvegardé.

1.2.2.1 Checkpoint pour un programme isolé

Attardons-nous d'abord au cas le plus simple où l'on cherche à sauvegarder l'état d'un programme exécuté par une seule machine pour le redémarrer en cas de panne.

Checkpoints de niveau système ou utilisateur

Si l'on souhaite ne pas modifier le code de l'application, on peut déléguer cette tâche au système d'exploitation, ou à une bibliothèque spécialisée. On parle alors de checkpoints de *niveau système* quand l'OS gère la sauvegarde d'un programme et de *niveau utilisateur* dans le cas d'utilisation de bibliothèques de checkpoint. On peut regrouper ces niveaux de checkpoints car ils utilisent une approche similaire. En effet le mécanisme de checkpoint ne peut ici faire aucune supposition sur le programme, ses données, les bibliothèques utilisées... il va devoir sauvegarder l'ensemble de l'application et son contexte afin de garantir l'intégrité du checkpoint. Cela implique les zones mémoires de l'application (statique, pile et tas), mais aussi celles des bibliothèques utilisées. D'autres données du système peuvent nécessiter d'être sauvées comme les descripteurs de fichiers ouverts ou encore les verrous acquis par l'application. Tout ceci représente rapidement des complexités logicielles importantes et des tailles de checkpoints conséquentes, qui vont se traduire par un coût en performance élevé. En effet lors de la création du checkpoint il faut mettre en pause l'exécution du programme pour s'assurer de l'intégrité des données ; si une partie des données changeait pendant la sauvegarde cela compromettrait le checkpoint, qui décrirait alors un état potentiellement incohérent, i.e. qui n'est pas

un état par lequel a pu passer l'application. On cherche à éviter cela car on ne pourrait plus garantir la capacité de fournir un résultat correct si l'on redémarrait le programme avec ce checkpoint incohérent. Les checkpoints doivent donc interrompre l'exécution du programme ce qui rallonge le temps de calcul, et il faut de plus à chaque checkpoint sauver l'ensemble du contexte de l'application.

Une fois le checkpoint terminé, le système peut si nécessaire restaurer l'ensemble de ces données dans la mémoire afin de redémarrer le programme. Si l'on doit redémarrer le programme sur une autre machine car celle d'origine est en panne, celle-ci doit être identique en configuration afin de garantir que restituer le contexte de l'application fonctionne. En effet une simple variation de version d'une des bibliothèques peut suffire à rendre impossible le redémarrage. Cette méthode est donc sujette à des problèmes de portabilité.

Un avantage de ce niveau cependant est que le checkpoint est dynamique : la bibliothèque ou le système crée le checkpoint de manière arbitraire, ce qui peut intervenir à n'importe quel moment du programme. Ainsi on ne fait aucune supposition sur le contenu de l'application et l'on peut choisir de créer des checkpoints selon une période optimale, qui correspond au minimum de l'overhead généré. Cette période optimale est facilement calculable avec le $MTBF$ du système et le temps que l'on met à effectuer un checkpoint d'après Young [67]. Ainsi les checkpoints au niveau entre système et utilisateur permettent d'effectuer des checkpoints dynamique de manière transparente, au prix d'une interruption de l'exécution et des sauvegardes volumineuses. Mais cette solution est en réalité la plus coûteuse en terme de temps et de ressources car elle ne peut faire aucune supposition sur le contenu des sauvegardes.

Checkpoints au niveau de l'application

À l'opposé, on peut choisir de traiter les checkpoints au *niveau de l'application*. Effectuer les sauvegardes à ce niveau a l'avantage de permettre au développeur de l'application d'indiquer quelles sont les données qu'il faut sauvegarder afin de s'assurer de l'intégrité des checkpoints. Sauvegarder uniquement les données importantes permet de réduire énormément la taille des checkpoints, mais un développeur doit en contrepartie être impliqué, là où les solutions précédentes étaient transparentes pour l'utilisateur. La complexité réside surtout dans le fait que la cohérence du redémarrage incombe au développeur de l'application ; ce dernier doit s'assurer qu'en cas de panne, l'ensemble des données soient réinitialisées dans un état correspondant à l'état dans lequel était le programme lors de la prise du checkpoint. Le code 1.2 montre le type d'interface que l'on retrouve habituellement dans les solutions de checkpoints au niveau application. Cet exemple simple met en évidence que l'on peut choisir de sauvegarder uniquement les données importantes du calcul, ici a et b, ce qui réduit la taille du checkpoint et

```c
int main(int argc, char* argv[]) {
    int a=0, b=0, i=0, i_0=0;

    // Indique les données à sauver lors du checkpoint.
    failuretolerance_protect(&a, &b, &i);

    // Est vrai si l'on redémarre après une panne.
    if failure_occured() {
        // On initialise d'après les données du checkpoint.
        a = last_checkpoint(&a);
        b = last_checkpoint(&b);
        i_0 = last_checkpoint(&i) + 1;
    }

    // On prend soin de reprendre à l'itération i_0 correspondant au checkpoint.
    for (i=i_0 ; i<IMAX ; i++) {
        a+=b;
        b++;

        // On sauvegarde les données (a, b et i).
        failuretolerance_checkpoint();
    }
}
```

Code 1.2 : Exemple de tolérance aux panne au niveau de l'application.

donc le coût en performance en comparaison d'un solution au niveau du système. Mais pour redémarrer conformément à l'exécution initiale, il faut également sauver la variable d'itération i et la restaurer avec la valeur du checkpoint lors du redémarrage. Dans cet exemple avec une structure itérative il est relativement simple de déterminer comment reprendre le code ; mais cet aspect peut rapidement être plus difficile à prendre en compte selon la structure de l'application.

On remarque également que lorsque l'on choisit de traiter les sauvegardes au niveau de l'application, les checkpoints doivent être insérés statiquement dans le code. Il est néanmoins possible d'automatiser les modifications que le code subit à l'aide d'un compilateur. Ce dernier peut analyser le code, déduire les données à sauvegarder et structurer une sauvegarde pour s'assurer que le code reprenne dans un état cohérent avec l'exécution initiale, en effectuant une *instrumentation de code* [47].

La principale différence entre le niveau application et les autres est représentée ici : on ne sait pas exactement à quel instant le checkpoint est exécuté dans le temps car cela dépend de l'avancement du programme, alors que l'on pouvait choisir de réaliser un checkpoint de manière arbitraire avec des checkpoints dynamiques. Ainsi, respecter la fréquence de checkpoint optimale de Young est plus compliquée à établir avec des checkpoints statiques, même s'il est possible d'anticiper quel est le temps d'exécution de différentes portions de code et de placer les checkpoints en conséquence. On peut faire tout de même en sorte de ne pas sauvegarder effectivement les checkpoints à chaque

Checkpoints	Niveau utilisateur	Niveau application
Flexibilité	Dynamique	Statique
Contenu	Tout le contexte	Uniquement donnée ciblées
Réinitialisation	Externe	Interne
Problèmes de portabilité	Sensible	Insensible
Modifications nécessaires	Non	Oui

TABLE 1.1 : Comparaison des niveaux utilisateur et application.

appel, en effectuant la sauvegarde du checkpoint uniquement si aucun checkpoint n'était fait pendant une période temporelle $t > T$, avec T une période de checkpoint minimale à respecter.

Une autre différence est que si le flux d'exécution est déterministe, on peut savoir avant l'exécution quel sera le contenu d'un checkpoint statique, alors qu'il est impossible d'anticiper le contenu d'un checkpoint dynamique.

Discussion

Pour résumer, ainsi que schématisé sur la table 1.1, les checkpoints de niveau système ou utilisateur permettent de sauvegarder à n'importe quel moment une application en créant un checkpoint contenant tout le contexte de l'application. La réinitialisation de l'application après panne est externe, c'est le système qui s'occupe de restituer l'ensemble de l'application et son contexte depuis toutes les données sauvegardées. En revanche avec un checkpoint de niveau application il est possible de sauver uniquement la sous-partie des données du programme qui est identifiée comme critique. En contrepartie il faut que l'application gère en interne la réinitialisation afin de se replacer dans un état cohérent avec l'exécution originelle. Le niveau application permet de réduire l'empreinte mémoire d'un checkpoint au prix de plus d'intrusivité dans le code, par rapport au niveau utilisateur. On peut noter que les checkpoints au niveau utilisateur peuvent être statiques (on peut très bien insérer dans le code une commande qui va demander à une bibliothèque ou au système de sauvegarder le contexte l'application) mais l'intérêt est limité. On pourrait également en théorie utiliser l'instrumentation de code pour effectuer des checkpoints dynamiques gérés au niveau de l'application, mais cela nécessite d'instrumenter entièrement le code de l'application ce qui est fastidieux et coûteux en performance. Dans ce qui a été présenté nous avions donc tendance à associer checkpoints statiques au niveau application, et checkpoints dynamiques au niveau utilisateur car ils sont historiquement liés mais en réalité compatibles entre eux. On s'attardera donc sur les spécificités des checkpoints statiques et dynamiques dans la suite de ce chapitre, sans se préoccuper du niveau d'implémentation.

Les approches présentées ne sont pas suffisantes, car comme évoqué en section 1.1,

la puissance des calculateurs vient du fait que chaque nœud effectue une partie du travail pour contribuer à l'avancement global du calcul. Ces nœuds doivent interagir afin de partager leurs contributions et progresser vers le résultat final. L'application étant distribuée nous sommes forcés de distribuer les checkpoints, chaque nœud s'occupant de faire sa propre sauvegarde. Toute la complexité du checkpoint distribué vient de ce fait, car l'on doit sauvegarder de manière locale l'état de chaque nœud tout en s'assurant que l'ensemble de ces checkpoints locaux représentent un état global cohérent [53], à savoir un état par lequel est passé l'application distribuée, ou par lequel elle aurait pu passer, et depuis lequel il est possible de terminer avec succès l'exécution.

On peut s'intéresser dans un premier temps aux solutions de checkpoints distribués dynamiques, qui ont l'avantage de proposer une intégration transparente et ne requièrent pas de modification du code de l'application. Puis l'on pourra voir comment les checkpoints statiques peuvent être une alternative intéressante, où la modification de code imposée par ces méthodes est compensée par les performances.

1.2.2.2 Checkpoints distribués dynamiques

La grande majorité des programmes distribués interagissent en utilisant des communications par messages telles que proposées par MPI, afin de distribuer leurs calculs sur plusieurs machines comme présenté en section 1.1. L'utilisation de MPI peut rendre l'exécution non déterministe, car les nœuds peuvent réaliser les opérations sur les données disponibles à un instant t, ce qui dépend de l'avancement des autres nœuds et de l'utilisation du réseau de communication. Ceci rend chaque exécution différente d'une autre du point de vue de l'ordre de progression parallèle du programme, même si cela ne remet pas en cause la reproductibilité du résultat final d'une exécution à l'autre. Certaines structures d'application permettent de faire des hypothèses quant au déterminisme du flux d'exécution [21], ce qui permet d'envisager de réaliser des checkpoints transparents de niveau utilisateur plus efficaces. Mais toute les classes d'applications ne permettent pas de supposer ces hypothèses et nécessitent d'utiliser des solutions plus générales qui sont évidemment plus coûteuses [66, 16, 2].

Sans hypothèses particulières, on doit donc créer pour chaque nœud un checkpoint local, formant une collection de checkpoints qui représente un état global cohérent, en considérant que les nœuds peuvent effectuer des communications à n'importe quel instant, de manière non déterministe.

La notion de sauvegarde *cohérente* est importante pour comprendre les problématiques liées aux checkpoints distribués, c'est pourquoi nous allons nous y attarder dans les prochains paragraphes.

Sur la Figure 1.4, on peut voir que les checkpoints locaux $C_{0,1}$, $C_{1,1}$ et $C_{2,1}$ ne forment pas un checkpoint global cohérent, car le message m_3 est considéré comme reçu

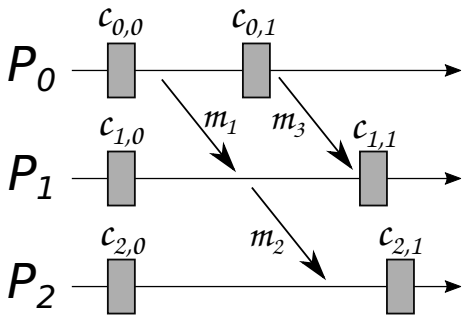


FIGURE 1.4 : Exemple de message *orphelin* : m_3 est considéré comme reçu dans $C_{1,1}$, mais comme non envoyé dans $C_{0,1}$.

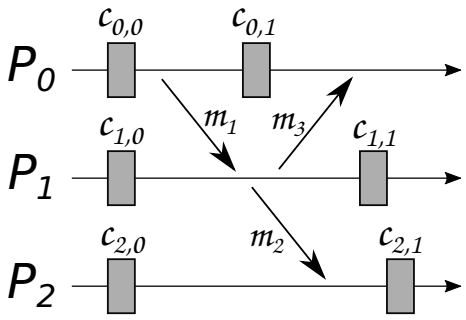


FIGURE 1.5 : Exemple de message *en transit* : m_3 est considéré comme non reçu dans $C_{0,1}$, mais comme envoyé dans $C_{1,1}$.

dans le checkpoint $C_{1,1}$ alors qu'il est considéré comme non envoyé dans le checkpoint $C_{0,1}$, ce qui est causalement impossible : on parle alors de message *orphelin*. Reprendre l'exécution en initialisant les nœuds sur ces checkpoints aboutirait à une incohérence complète car cela ne représente pas un état par lequel a pu passer l'application, ce qui va forcément forcément impacter le résultat du calcul et remettre en question la validité : le nœud P_1 ne peut pas avoir reçu le message m_3 si le nœud P_0 ne l'a pas encore envoyé. Si l'on redémarre l'exécution d'après ces checkpoints, le message m_3 sera reçu une seconde fois par le nœud P_1 lorsqu'il attendra le prochain message en provenance du nœud P_0 ce qui va corrompre son état et créer une incohérence.

On peut également voir un autre type de messages problématiques sur la Figure 1.5. Les checkpoints $C_{0,1}$, $C_{1,1}$ et $C_{2,1}$ ne peuvent pas être utilisés car le message m_3 est considéré comme non reçu dans le checkpoint $C_{0,1}$ alors qu'il est considéré comme envoyé dans le checkpoint $C_{1,1}$, c'est un message dit *en transit*. Reprendre l'exécution à partir de ces checkpoints aboutirait à un manque de données : P_1 ne renverra pas le message m_3, qui devient alors un message *perdu*.

C'est l'un des points principaux sur lequel s'attardent les propositions d'algorithmes de checkpoints distribués afin de respecter la cohérence de l'état global, à savoir s'assurer de l'absence de ce type de messages orphelins ou en transit.

Pour ne pas avoir ce cas de figure, on va chercher à limiter les interactions entre les

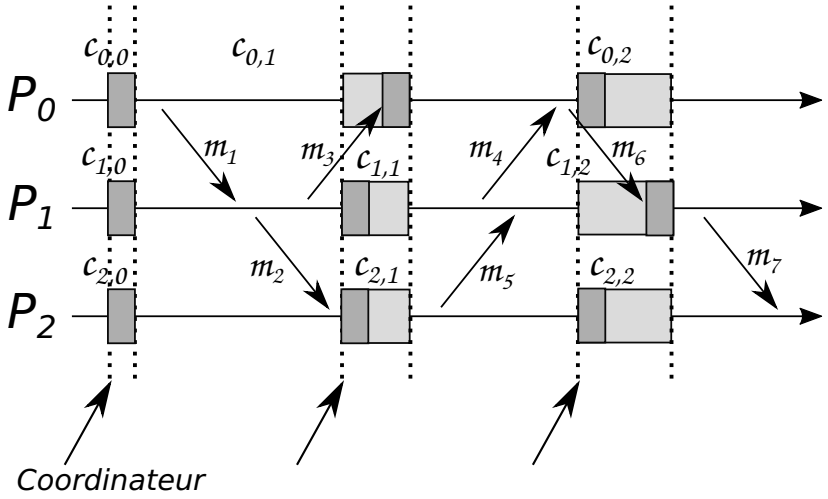


FIGURE 1.6 : Exemple de checkpoints coordonnés bloquants : chaque nœud doit éventuellement attendre (en gris clair) la réception des messages et que les autres nœuds aient fini leur checkpoint.

nœuds lorsque l'on crée un checkpoint distribué. On peut distinguer deux approches pour implémenter des checkpoints distribués, à savoir réaliser la sauvegarde de manière coordonnée ou non.

Checkpoints coordonnés

Comme l'on souhaite limiter les interactions entre les nœuds pendant une sauvegarde, un moyen d'y arriver est tout simplement d'empêcher ces interactions. La façon la plus évidente est d'avoir un coordinateur qui interrompt l'ensemble des instances du programme distribué en invitant chaque nœud à réaliser son checkpoint, et ne les laisse reprendre leur exécution qu'une fois tous les checkpoints effectués. Cela est visible sur la Figure 1.6.

En considérant que la couche de communication l'autorise, un processus interrompu par une requête de checkpoint doit forcer la réception de tous les messages en attente afin de ne pas avoir de message *en transit*, à savoir un message qui a été émis avant une sauvegarde mais pas encore reçu. On peut voir sur la figure 1.6 que le message m_3 est en transit lors de la requête de checkpoint, sa réception est donc forcée avant la sauvegarde car il ne sera pas réémis en cas de sauvegarde, devenant un *message perdu*. À noter que ceci présuppose que la gestion de la fiabilité des communications ne fait pas partie de ce que l'on doit sauvegarder en cas de panne, auquel cas un tel message perdu serait traité comme n'importe quel erreur de transmission lors d'une exécution sans panne. Dans ces conditions on ne peut pas avoir de message orphelin car l'on empêche toute rupture de causalité. Néanmoins le programme étant entièrement interrompu à chaque

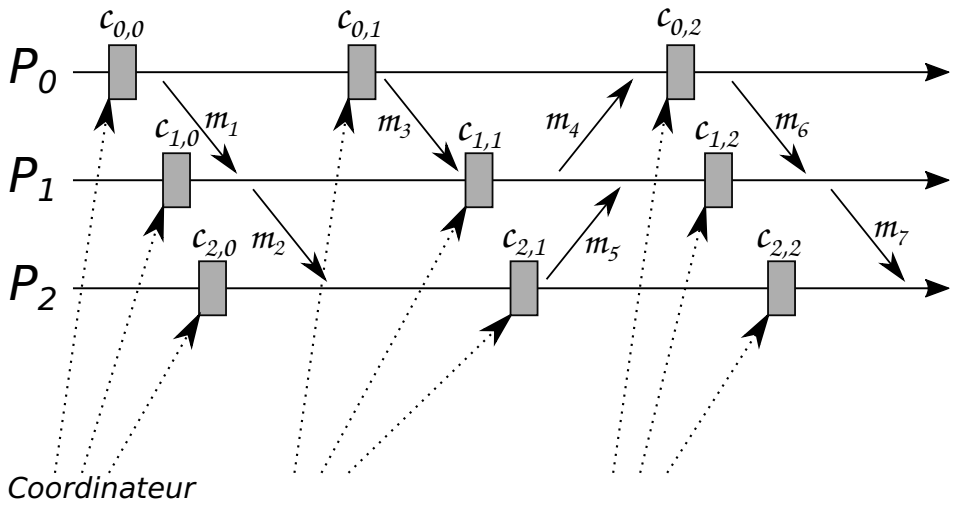


FIGURE 1.7 : Exemple de checkpoints coordonnés non bloquants : le coordinateur contacte chaque nœud un par un. Pour conserver une cohérence, il faut retarder la réception du message m_3 après le checkpoint $C_{1,1}$.

sauvegarde, l'ensemble des nœuds attendent que le plus lent complète son checkpoint, ce qui engendre bien évidemment une dégradation des performances considérable.

Cependant un checkpoint coordonné n'est pas nécessairement bloquant [19]. Pour réaliser des checkpoints coordonnés de manière non-bloquante, un coordinateur peut contacter chaque nœud un par un pour initier un checkpoint. Mais en l'absence de synchronisation on ne peut pas garantir la simultanéité des checkpoints, certains nœuds pouvant potentiellement communiquer lors d'une sauvegarde et créer une incohérence dans l'état décrit par le checkpoint global. Cette incohérence est induite une nouvelle fois par la présence de messages orphelins, qui sont reçus mais non envoyés dans l'état décrit par le checkpoint, comme illustré pour le message m_3 sur la Figure 1.7. Il faut dans ce cas retarder la réception de tels messages, en ajoutant dans ce message une horloge de Lamport permettant à l'émetteur d'indiquer qu'il a effectué un checkpoint que le récepteur n'a peut être pas encore fait, et ainsi forcer une sauvegarde avant d'effectuer la réception de ce message si le récepteur est effectivement en retard ; c'est ce qui est proposé par l'algorithme Chandy-Lamport dans [26]. À noter que les mêmes conditions que pour le checkpoint coordonné bloquant s'appliquent au traitement des messages en transit.

Checkpoints non coordonnés

Pour réaliser des checkpoints de manière non coordonnée, les nœuds vont chacun prendre la liberté d'effectuer des checkpoints de manière indépendante. En rajoutant des messages de contrôle, un nœud va pouvoir informer les autres à propos de son dernier checkpoint.

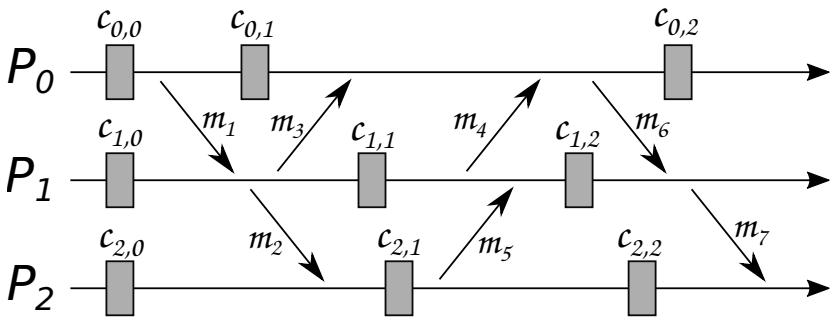


FIGURE 1.8 : Exemple de checkpoints non coordonnés, m_3 et m_6 sont en transit.

Lors d'une panne, il est possible de récupérer l'information de ces messages de contrôle afin de déterminer quel jeu de checkpoints correspond à un état global cohérent, et redémarrer l'application depuis cet état. Néanmoins l'existence d'un tel état n'est pas assurée, et il est possible que le seul état global cohérent soit l'état initial : c'est l'*effet domino*, apportant le risque que le mécanisme de checkpoints soit finalement inutile. La Figure 1.8 met en évidence la présence de messages en transit et la non trivialité d'avoir un checkpoint global cohérent. Dans cet exemple les trois processus effectuent des checkpoints non coordonnés, sans s'influencer les uns les autres. Si le nœud hébergeant l'un des processus tombe en panne, les processus restants déterminent en consensus quel est le dernier jeu de checkpoints globalement cohérent. Les checkpoints $C_{0,2}$, $C_{1,2}$ et $C_{2,2}$ ne peuvent pas former un point de reprise cohérent car le message m_6 est alors en transit. En continuant de prospecter pour un ensemble de checkpoints globalement cohérent, on se rend compte que l'on recule de plus en plus et que l'on subit un effet domino. Dans cet exemple le seul jeu de checkpoint candidat à la reprise est le jeu de checkpoint initial, ce qui neutralise l'utilité des sauvegardes. On peut percevoir dans cet exemple que faire davantage de checkpoints augmenterait la probabilité de trouver un point de reprise cohérent, mais sans le garantir. Même si les checkpoints non coordonnés ne donnent pas lieu à des barrières sur l'ensemble des nœuds, chaque processus doit néanmoins être interrompu le temps du checkpoint et les coûts en performance sont donc conséquents, tandis que l'utilité d'un checkpoint n'est pas assurée. En revanche il est possible de faire des suppositions sur les caractéristiques de certaines applications et de compléter la solution pour éviter un effet domino, comme la proposition de Guermouche et al. [43], qui s'appuie sur le déterminisme partiel de la plupart des applications [21] et sur la sauvegarde des derniers messages pour former un *log de messages* [4]. Il n'existe cependant pas de solution de checkpoint non coordonné raisonnable et aussi simple qui permette de garantir la tolérance aux pannes pour l'ensemble des classes d'applications.

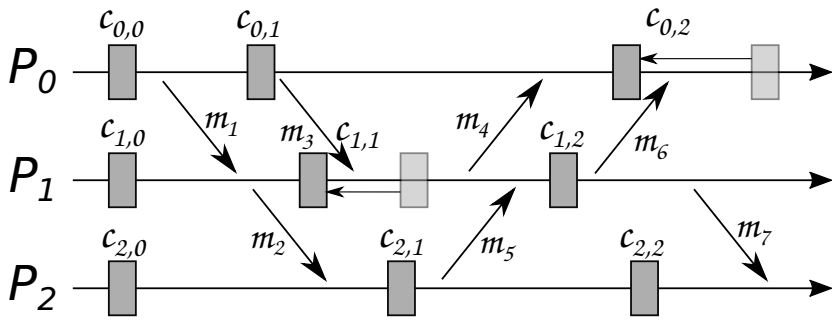


FIGURE 1.9 : Exemple de checkpoints induits par communication : la réception du message m_3 impose à P_1 de produire le checkpoint $C_{1,1}$ avant de réellement réceptionner ce message m_3 ; de même pour m_6.

Checkpoints induits par communication

Une proposition particulière des checkpoints non coordonnés est la solution des *checkpoints induits par communication*. Ceux-ci proposent de laisser la liberté à chaque nœud de réaliser un checkpoint *local* selon des critères propres au nœud, mais aussi d'être influencé par les checkpoints réalisés par les autres nœuds. Dans le checkpoint non coordonné évoqué précédemment, on a pu voir que certains checkpoints ne servent à rien car ils n'est pas garanti qu'ils appartiennent à un checkpoint global cohérent.

Le but de cette nouvelle approche va être de s'assurer qu'aucun checkpoint local ne soit inutile, en poussant les autres nœuds à effectuer un checkpoint forcé par un mécanisme détectant les *zigzag paths* [59, 53] qui est une évolution des horloges de Lamport. Par exemple sur la Figure 1.9, lorsque P_1 reçoit le message m_3, il doit réaliser le checkpoint $C_{1,1}$ avant de réellement réceptionner m_3, pour éviter que celui-ci devienne un message en transit. Étant donné qu'il n'y a pas de coordinateur, plusieurs nœuds peuvent choisir de créer un checkpoint local dans un même intervalle de temps, initiant chacun une *vague de checkpoints* qui évolue simultanément. De plus il est possible de détecter qu'un checkpoint est potentiellement inutile, mais sans le garantir. L'algorithme permet de garantir qu'un checkpoint local fasse partie d'un checkpoint global cohérent et donc assure qu'il ne soit pas inutile, en forçant les autres à faire un checkpoint si nécessaire.

Néanmoins les algorithmes existants surestiment le nombre de checkpoints forcés nécessaires, car obtenir les informations pour déterminer le nombre exact de checkpoints forcés nécessite de nombreuses communications entre les nœuds, ce qui va à l'encontre de l'idée de base de cette méthode. Cette approche est efficace car elle garantit la présence de checkpoints globaux cohérents sans synchronisation globale et sans ajouter de communications spécifiques au protocole de tolérance aux pannes (on utilise les communications de données du programme pour partager les informations nécessaires).

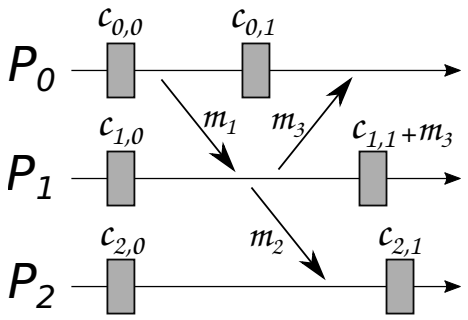


FIGURE 1.10 : Exemple de log de message : P_1 sauvegarde le message m_3 en plus du checkpoint $C_{1,1}$, pour pouvoir être prêt à le réémettre si P_0 était obligé de reprendre l'exécution depuis le checkpoint $C_{0,1}$.

En revanche ceci fait que l'information locale à propos des checkpoints réalisés par le reste du système est limitée, et l'on réalise en pratique beaucoup plus de checkpoints que nécessaire ce qui crée une perte de performance évitable par d'autres approches [35].

Message logging

On retrouve de nombreuses approches qui proposent de compléter un mécanisme de checkpoint avec un *log de messages* [4, 18]. Le principe, illustré à la Figure 1.10, est de sauvegarder une partie des messages afin de les rejouer dans le cas d'un redémarrage après panne, dans le même ordre que l'exécution initiale dans un souci de cohérence. Il offre des informations supplémentaires qui permettent d'affiner les protocoles présentés précédemment [43, 3]. C'est une solution intéressante pour évacuer le problème évoqué plus tôt des messages en transit lors d'un checkpoint qui sont perdus en cas de panne et les messages orphelins. Elle permet de compléter l'approche des checkpoints non-coordonnés, afin de résoudre le problème de cohérence de checkpoints. Néanmoins sauver tous ces messages provoque souvent une empreinte mémoire de sauvegarde énorme, mais dans un contexte où l'ordre d'envoi des messages est déterministe il est possible de réduire cette empreinte comme l'on montré Guermouche et al. [43]. C'est également une solution qui autorise de ne redémarrer que le nœud en panne, ce dernier étant alors initialisé au dernier checkpoint avant que les autres nœuds ne rejouent les messages afin de le replacer dans un état cohérent avec l'ensemble du réseau. Il y a plusieurs manières d'envisager l'implémentation du log de message [4] selon les suppositions faites sur l'application et l'infrastructure. Il n'est pas utile de s'attarder sur cette approche ici car nous nous repencherons dessus longuement en Section 3.2.

```c
#include <mpi.h>
int main() {
    // Les 2 nœuds exécutent le même programme (SPMD)
    MPI_Init();
    ... //Init
    ... //Work
    if (rank == 0) {
        // Exécuté uniquement par le nœud 0
        MPI_ISend(&a, 1/*node*/, &req);
        MPI_Recv(&b, 1/*node*/);
    }
    ... //Work
    // On sauvegarde les données.
    checkpoint();
    ... //Work
    if (rank == 1) {
        // Exécuté uniquement par le nœud 1
        MPI_ISend(&b, 0/*node*/);
        MPI_Recv(&a, 0/*node*/);
    ...
    }
    MPI_Wait(&req);
    // Point de synchronisation
    MPI_Barrier(MPI_COMM_WORLD);
    // Terminaison de MPI
    MPI_Finalize();
}
```

Code 1.3 : Exemple de checkpoints statiques incohérents.

1.2.2.3 Checkpoints statiques

Les solutions évoquées précédemment sont conçues pour réaliser des checkpoints dynamiques de manière transparente à l'application, ne requérant aucune implication du développeur de l'application pour la rendre tolérante aux pannes, ce qui impose les limites évoquées. Mais en injectant plus d'informations dans le code de l'application il est possible de simplifier davantage la problématique. Comme évoqué en 1.2.2.1, le checkpoint peut être inséré statiquement dans le code de l'application. En bénéficiant des propriétés SPMD des programmes MPI, on peut s'assurer dans une certaine mesure de la cohérence des checkpoints.

Insérer un checkpoint statiquement dans le code d'une application SPMD permet d'obtenir une synchronisation logique entre tous les nœuds, sans pour autant nécessiter une synchronisation physique ; les checkpoints étant insérés au même endroit pour tous les nœuds, chaque nœud est assuré que les autres nœuds font leurs checkpoints au même endroit dans la file d'exécution. Mais les relations de causalités ne sont pas forcément respectées et impose une rigueur d'écriture. Pour garantir la cohérence il faut tout de même s'assurer que l'émission et la réception d'un même message ne soient pas séparées par un checkpoint, car sinon on aurait des messages *orphelins* ou *en*

transit (Section 1.2.2.2).

L'exemple de code 1.3 permet de mettre en évidence ce problème, dans lequel on précise les données. Le nœud 0 attend la donnée b en provenance du nœud 1, puis réalise son checkpoint, alors que le nœud 1 crée son checkpoint avant d'envoyer cette donnée : le checkpoint global créé est alors incohérent car il enregistre que la donnée b est reçue mais non envoyée, et l'on a donc un message orphelin. Inversement, la donnée a est envoyée par le nœud 0 avant le checkpoint, mais reçue par le nœud 1 après le checkpoint : on a donc un message en transit et il sera perdu s'il n'est pas réémis au redémarrage. Le standard MPI autorise de créer des applications qui correspondent à ce schéma, et ces incohérences doivent être considérées. Néanmoins le problème est évacué si le développeur prend garde à ne pas séparer par un checkpoint une émission et une réception correspondante. La plupart des applications peuvent faire cela facilement mais il est compliqué de respecter cette condition si l'on entrelace fortement les communications et le calcul ; il peut être alors nécessaire de prendre en compte cette particularité par un autre moyen si l'on ne souhaite pas imposer au développeur de l'application cette rigueur. Aussi, on remarque que l'écriture du checkpoint est bloquante au niveau du nœud, pendant laquelle on ne réalise aucun calcul jusqu'à ce que la sauvegarde locale soit complète. Dans les contributions que nous proposons en partie 2.1.1.1, nous verrons que non seulement nous garantissons la cohérence globale des checkpoints quel que soit leur placement dans l'application, mais également que l'on autorise le calcul à être poursuivi en parallèle de la sauvegarde.

On peut noter aussi que les checkpoints statiques peuvent être gérés au niveau de l'application, et il est possible de laisser l'utilisateur spécifier quelles données doivent être sauvegardées. Il faut cependant que le code d'application gère aussi la reprise après panne, comme présenté en Section 1.2.2.1. Les outils FTI [10] ou encore VeloC [54] fonctionnent de cette manière, et proposent en plus d'envoyer les sauvegardes sur plusieurs supports de stockage, avec différentes fréquences en fonction de la latence induite par le support et en fonction de la probabilité de perdre ce support en cas de panne.

On peut noter aussi que dans ces approches, reprendre l'exécution au bon endroit est à la charge de l'utilisateur. En effet il faut reprendre l'exécution du programme à un endroit précis, alors que l'on a uniquement les données des checkpoints. Il faut donc écrire un code qui puisse reprendre à une instruction précise (qui est celle qui est juste après le checkpoint utilisé) en fonction de données de contrôle, et sauver ces données de contrôle dans le checkpoint. Le code 1.4 a été instrumenté avec des données de contrôle afin de pouvoir reprendre l'exécution au bon endroit. Dans ce code on sauve une variable `i_0` qui reflète la variable d'itération `i`, ainsi qu'une donnée de contrôle de progression `p`, afin de reprendre l'exécution au bon endroit en se basant sur un

```c
#include <mpi.h>
int main() {
    // Les 2 nœuds exécutent le même programme (SPMD)
    MPI_Init();
    int i, i_0, p, a, b;
    i_0 = 0;
    p = 0;
    checkpoint_protect(i_0, p, a, b);
    if checkpoint_available() {
        restore(i_0, p, a, b);
    }
    for(i = i_0 ; i<N ; i++) {
        if(0==p) {
            ... // Work with data a and b and communications
            p = 1;
            checkpoint();
        }
        if(1==p) {
            ... // Work with data a and b and communications
            p = 0;
            i_0 = i+1;
            checkpoint();
        }
    }
    // Point de synchronisation
    MPI_Barrier(MPI_COMM_WORLD);
    // Terminaison de MPI
    MPI_Finalize();
}
```

Code 1.4 : Instrumentation de code pour atteindre un point de reprise.

checkpoint. Cette instrumentation peut être vue comme une lourdeur pour l'utilisateur, mais les performances offertes par cette approche sont très intéressantes en comparaison des checkpoints niveau système, qui eux sont certes transparents pour l'utilisateur, mais dégradent énormément les performances.

1.2.2.4 Lieu de stockage des checkpoints

Dans la plupart des solutions de tolérance aux pannes évoquées jusqu'à présent, un point d'honneur est mis sur la quête de la fiabilité, où l'on cherche à garantir l'exécution quelle que soit la panne subie, sans se soucier de la portée de celle-ci et de sa probabilité d'occurrence. De manière générale on cherche à sauvegarder les checkpoints sur *stockage stable* qui est considéré fiable et immunisé aux pannes. En pratique il s'agit du système de fichiers du super-calculateur (PFS : Parallel File System) qui intègre un mécanisme de tolérance aux pannes qui lui est propre, afin de garantir la disponibilité des données qui y sont stockées. Un problème inhérent à travailler ainsi est que la bande passante du PFS risque la saturation lorsque les checkpoints y sont déposés, augmentant significativement le coût lors de la sauvegarde. Néanmoins la nécessité d'utiliser le PFS pour y stocker les sauvegardes n'est pas justifié dans la majorité des scénarios de panne. Il peut être par exemple pertinent de stocker les sauvegardes dans la mémoire locale des nœuds afin de s'assurer de couvrir les pannes les plus probables. Il a été par exemple proposé de stocker le checkpoint d'un nœud dans la mémoire d'un nœud voisin (appelé *buddy*), afin de bénéficier de la localité du réseau et d'éviter une saturation de bande passante [68, 69, 31, 62]. Néanmoins ne pas sauver les checkpoints sur stockage stable impose de considérer le cas où le nœud d'origine et l'hôte de la sauvegarde tomberaient tous deux en panne en même temps. Même s'il existe des corrélations entre les pannes [45, 49] et le fait que deux nœuds puissent être rendus inopérants par la même cause, la probabilité d'occurrence de ces scénarios est plus faible que la probabilité d'avoir une panne n'affectant qu'un seul nœud [58]. Procéder ainsi permet de surmonter les pannes les plus courantes, et le risque de tomber sur une panne que l'on ne peut gérer peut être considéré acceptable si le coût en performance induit par le mécanisme de tolérance aux pannes est nettement inférieur au coût imposé par un mécanisme qui saurait gérer ces scénarios. C'est en partant de cette approche qu'ont été développées des solutions de *checkpoint multi-level* comme FTI [10] ou encore VeloC [54], permettant de sauvegarder plus ou moins fréquemment selon le niveau de mémoire hébergeant la sauvegarde, en fonction de la vulnérabilité de la zone mémoire et du coût en performance de la sauvegarde [30].

1.2.2.5 Checkpoint différentiel et Checkpoint incrémental

Il est possible d'effectuer un *checkpoint différentiel* [48, 52, 64], qui propose de sauver les checkpoints de manière différentielle, à savoir ne sauver que les sous-parties de checkpoint qui diffèrent des checkpoints précédents ou de l'état initial, réduisant ainsi la bande passante consommée pour sauver les données. Cette approche permet de réduire de manière significative le coût de transfert et de stockage des checkpoints pour les applications dont l'état évolue peu entre les checkpoints.

Le *checkpoint incrémental* est une approche qui permet de sauver les checkpoints par morceaux, en initiant la sauvegarde d'une sous-partie du checkpoint dès qu'elle est prête sans pour autant qu'elle ne constitue un checkpoint complet, permettant d'étaler la sauvegarde dans le temps et de diminuer la contention sur les supports de stockage. Les systèmes faisant des checkpoints de manière synchrone peuvent bénéficier de cette approche en faisant une copie des données locales et de reprendre le calcul avant de commencer à envoyer au support mémoire les sous-parties du checkpoint de manière diluée dans le temps. Cela permet d'étaler la bande passante consommée pour transférer les données, mais retarde par conséquent la terminaison du stockage d'un checkpoint. Comme indiqué dans [48], on observe une confusion de nomenclature dans la littérature, où le terme de checkpoint incrémental est souvent utilisé pour désigner le checkpoint différentiel, ou les deux approches combinées.

1.2.3 Redémarrer le calcul

Nous avons évoqué jusqu'à présent principalement le redémarrage global. En effet les efforts de définition de checkpoints globaux cohérents sont effectués dans cette optique. Cette approche est plus simple à mettre en pratique qu'un redémarrage local, cette autre approche faisant en sorte que les nœuds qui n'ont pas subi de panne continuent d'exécuter leur calcul, ce qui leur impose des contraintes supplémentaires. Néanmoins si l'on peut l'implémenter sans overhead supplémentaire conséquent, le redémarrage local permet d'être plus efficace étant donné que l'on réduit la quantité de calcul perdu. On pourra aussi mentionner une autre approche qui s'intéresse à continuer le calcul avec un nombre réduit de nœuds.

1.2.3.1 Redémarrage global

Cette approche consiste à redémarrer l'ensemble des nœuds de calcul, et permet de reprendre l'exécution indépendamment du nombre de machines tombées en panne. Chaque nœud doit donc reprendre son exécution. Pour des checkpoints de niveau utilisateur ou système, il n'est pas facile de reconstituer l'ensemble des nœuds du point de vue de MPI. Des implémentations de MPI comme MVAPICH2 [25] proposent

des solutions, mais aucune n'est en phase d'être ajoutée au standard MPI. Pour des checkpoints de niveau application des travaux ont été historiquement réalisés [37, 38], mais de nouveaux candidats au standard se présentent depuis quelques années. On peut citer ULFM [1], qui devrait être disponible dans de futures versions stables des implémentations OpenMPI [41] et MPICH [13]. Cette solution permet de réaliser les trois types de reprises présentées ici (global, local et shrink), mais peut manquer d'efficacité sur le redémarrage global [40, 44] face à un autre candidat : Reinit[24]. L'approche de Reinit se focalise uniquement sur le redémarrage global, et est particulièrement pensée pour des applications bulk-synchrones en permettant une meilleure efficacité et une meilleure prise en main qu'ULFM.

1.2.3.2 Redémarrage local

Il peut être intéressant de redémarrer uniquement le nœud en panne, et de faire en sorte que les nœuds non affectés par la panne continuent leur exécution normalement. Ainsi moins de calcul est perdu. Beaucoup de solutions de checkpointing ne s'intéressent pas à cette approche étant donné que l'on perd la cohérence établie avec les checkpoints globaux. Ces incohérences doivent être prises en considérations en sauvant davantage de données, ce qui crée bien souvent un overhead plus grand que le bénéfice obtenu en ne redémarrant qu'un seul nœud. Mais lorsqu'il est possible d'utiliser cette approche avec un overhead maîtrisé et acceptable, son utilisation est justifiée. Mais l'implémentation MPI doit pouvoir supporter cette stratégie de redémarrage, et ULFM propose précisément d'ajouter au standard quelques fonctionnalités permettant d'atteindre cet objectif. C'est le choix de rigueur si l'on souhaite effectuer des checkpoints de niveau application avec redémarrage local.

1.2.3.3 Shrink : continuer sur un nombre réduit de nœuds

Si l'application permet d'être modulable facilement quant au nombre de nœuds qui exécutent le calcul, on peut faire en sorte que la charge du nœud perdu soit répartie sur les nœuds restants. Du point de vue de MPI, on réduit le communicateur [5] au nombre de nœuds restants, et ULFM permet de réaliser cette approche.

1.2.4 ULFM

Le projet ULFM [12] propose une extension au standard MPI, afin de pouvoir rétablir les capacités de communication entre les nœuds survivants. Cette extension propose un mécanisme de détection de pannes [15] et de faire remonter l'information de pannes à l'application en remontant de nouvelles erreurs MPI spécifiquement définies. Également ULFM fournit de nouvelles primitives permettant de manipuler les communicateurs

affectés par une panne, et d'en constituer de nouveaux où les nœuds en panne ont été enlevés. À partir de là on peut choisir soit de remplacer le nœud perdu, soit de continuer l'exécution avec un nombre réduit de nœuds. L'interface proposée est assez simple mais en même temps très puissante, permettant de traiter une panne de la manière souhaitée par l'utilisateur, en ajoutant seulement quelques fonctions au standard MPI. S'il est possible d'effectuer du redémarrage global avec ces fonctionnalités, on préférera dans ce cas utiliser la solution proposée par Reinit [24] qui se montre plus adaptée. Mais pour effectuer un redémarrage local ou ne continuer qu'avec un nombre réduit de nœuds, cette solution est ce qui se propose de mieux si l'on souhaite faire des checkpoints de niveau application.

1.2.5 Bilan

On a vu qu'intégrer un mécanisme de tolérance aux pannes dans un programme distribué n'est pas une chose triviale si l'on souhaite réduire l'impact sur les performances. En effet il faut que la sauvegarde complète des différents nœuds représente un état cohérent du calcul. On a pu distinguer deux approches pour effectuer des checkpoints :

— des checkpoints de niveau système ou bibliothèque niveau utilisateur, permettant de sauvegarder de manière transparente à l'utilisateur une application et son contexte afin de restituer l'exécution d'un processus. Néanmoins cette approche doit utiliser une implémentation MPI adaptée, et n'est pas suffisamment efficace car la taille des checkpoints est trop grande, générant un overhead significatif. De plus la gestion de la cohérence des checkpoints est problématique et nécessite soit des coordinations, soit des algorithmes complexes qui induisent davantage de checkpoints que ceux exploitables en pratique.

— des checkpoints de niveau application, qui nécessitent un soin particulier de la part de l'utilisateur en les insérant statiquement dans son application. Si l'application est SPMD, on a une coordination implicite des nœuds, mais certaines règles doivent être respectées afin de s'assurer que chaque checkpoint local appartient à un checkpoint global cohérent. Le fait d'impliquer l'utilisateur donne en réalité l'opportunité d'identifier les données qui représentent vraiment l'état de l'application, permettant de réduire considérablement la taille des checkpoints.

Choisir l'emplacement d'une sauvegarde est également important, et il faut trouver un équilibre entre le coût de la sauvegarde sur ce support et la fiabilité du support. On peut utiliser dans ce cas des solutions qui sauvegardent sur différents supports à différentes fréquences afin d'avoir une meilleure efficacité. Le choix d'effectuer un redémarrage local apporte davantage de contraintes techniques qu'un redémarrage global, mais les bénéfices obtenus sont intéressants.

L'éventail des possibilités offertes par la littérature est assez large et on pourra voir au Chapitre 2 comment mettre en commun les propriétés de StarPU avec les différentes approches qui ont été présentées dans cette section.

1.3 StarPU : support d'exécution distribué à base de tâches

StarPU [6] est un *support d'exécution* (*runtime* en anglais), qui a pour but de faciliter l'exploitation des ressources de calcul au développement de l'application. Ce runtime adopte un modèle de programmation à base de tâches qui diffère des habitudes conventionnelles de programmation, mais qui ne demande que peu d'adaptations et reste très lisible. Ce modèle de tâches permet au développeur de se décharger du souci de l'exploitation des ressources, qui est réalisé automatiquement par StarPU. L'idée derrière la programmation à base de tâches est que chaque tâche est l'équivalent d'un appel de fonction, qui possède des données d'entrée et de sortie. StarPU sait ainsi quelles données sont accédées par les tâches, et peut en déduire un graphe de tâches dont les dépendances sont issues des accès des tâches aux données et de l'ordre dans lequel les tâches sont écrites dans le code source. Ce support d'exécution permet également de proposer, pour la fonction d'une tâche, plusieurs implémentations, chacune étant spécifique à un type de processeur (CPU, GPU, FPGA, ...). Le principe est alors de laisser un ordonnanceur décider si une tâche doit être exécutée sur CPU ou GPU par exemple. Il est également possible d'utiliser un mode distribué qui permet à une application d'être exécutée selon le principe SPMD. Pour ce faire StarPU utilise MPI, et déduit automatiquement les transferts de données nécessaires à l'application. L'aspect le plus intéressant de StarPU pour la tolérance aux pannes est la façon dont les tâches sont décrites et comment le graphe de tâches en est déduit.

1.3.1 Modèle STF de soumission des tâches

StarPU permet à l'application de décrire le calcul à effectuer avec des tâches, où chaque tâche consomme des données d'entrée, exécute un calcul, puis produit des données en sortie. Ainsi chaque tâche peut être considérée comme une boîte noire qui lit des données et écrit d'autres données. Dans le modèle de programmation STF (Sequential Task Flow), l'application soumet les tâches dans un ordre séquentiel (qui est en fait un parcours topologique du graphe de tâches à soumettre). Par exemple sur la Figure 1.11b, une première tâche est soumise qui produit en sortie une nouvelle valeur de la donnée A. La deuxième tâche soumise consomme cette nouvelle valeur, introduisant ainsi une dépendance, et induisant ainsi le graphe de tâches de la Figure 1.11c.

Cette soumission séquentielle des tâches permet d'exploiter une propriété de causalité, de savoir dans quel ordre sont modifiées les données. C'est grâce à cela que l'on

peut déduire les dépendances entre les tâches et que l'on peut effectuer des tâches indépendantes en parallèle. C'est ainsi que l'on obtient un résultat différent si l'on soumet d'abord une tâche 1 modifiant une donnée A puis une tâche 2 modifiant la même donnée A (voir Figure 1.11b), ou bien que l'on soumet au contraire d'abord une tâche 2 modifiant une donnée A puis une tâche 1 modifiant la même donnée A 1.11d. Une dépendance apparaît effectivement sur la donnée A entre les deux tâches, l'ordre d'écriture dans le code détermine la causalité. Comme on peut voir sur les figures 1.11c vs 1.11e, les graphes de tâches sont différents.

Par contre, soumettre d'abord une tâche 1 modifiant une donnée A puis une tâche 2 modifiant une autre donnée B (voir Figure 1.11f), est équivalent à soumettre d'abord la tâche 2 modifiant la donnée B puis la tâche 1 modifiant la donnée A (voir Figure 1.11h), car les tâches sont indépendantes. Le graphe déduit est effectivement le même dans les deux cas : les graphes des figures 1.11g et 1.11i sont équivalents.

Pour que StarPU puisse déduire ces dépendances ainsi, le modèle STF impose que les données soient gérées uniquement par StarPU et qu'elles ne soient pas modifiées autrement que par des tâches. StarPU propose pour cela la notion de *data handle*, qui est une structure permettant de manipuler une donnée et de suivre les modifications qu'elle subit. Dans le code de soumission de tâches de la Figure 1.11, `data_A` et `data_B` sont en fait des références à deux data handles, qui représentent les données A et B, qui ont été enregistrés au préalable (voir Figure 1.11a).

1.3.1.1 Exécution asynchrone

Il est important de comprendre que dans un support d'exécution à base de tâches, on distingue *soumission* et *exécution* des tâches.

Le thread principal de l'application effectue typiquement de nombreux appels à `starpu_task_insert` qui sont non-bloquants : ils ne font que soumettre les tâches au moteur d'exécution, qui en infère un graphe de tâches. Le thread principal soumet ainsi rapidement l'ensemble du graphe de tâches à exécuter, sans avoir à attendre qu'elles s'exécutent réellement.

Le moteur d'exécution, en parallèle, détermine automatiquement quelles tâches sont prêtes (celles qui n'ont pas de dépendances entrantes), et quelles tâches doivent encore attendre (car elles ont des dépendances sur des tâches non terminées). Il distribue alors les tâches prêtes aux différentes ressources de calcul disponibles (cœurs CPU, stream GPU, etc.).

Cette distinction entre soumission et exécution est importante car c'est ce qui permet au moteur d'exécution de traiter des tâches indépendantes en parallèle. Par exemple, avec le code applicatif de la Figure 1.11f, le moteur n'a pas obligation d'attendre la fin de la tâche 1 avant de commencer à exécuter la tâche 2, puisqu'elles sont indépendantes.

```
starpu_vector_data_register(&data_A      /*Handle pour la donnée A*/,
                            STARPU_MAIN_RAM,
                            A,           /*Pointeur vers la donnée A*/,
                            n,           /*Nombre d'éléments de la donnée A*/,
                            sizeof(int) /* Taille d'un élément */);
starpu_vector_data_register(&data_B      /*Handle pour la donnée B*/,
                            STARPU_MAIN_RAM,
                            B,           /*Pointeur vers la donnée B*/,
                            n,           /*Nombre d'éléments de la donnée B*/,
                            sizeof(int) /* Taille d'un élément */);
```

(a) Enregistrement des handles.

```
starpu_task_insert(&cl1, STARPU_RW, data_A, 0);
starpu_task_insert(&cl2, STARPU_RW, data_A, 0);
```

(b) Soumission de la tâche 1 puis la tâche 2 sur la même donnée A.

(c) Graphe produit.

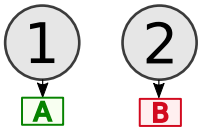
```
starpu_task_insert(&cl2, STARPU_RW, data_A, 0);
starpu_task_insert(&cl1, STARPU_RW, data_A, 0);
```

(d) Soumission de la tâche 2 puis la tâche 1 sur la même donnée A.

(e) Graphe produit.

```
starpu_task_insert(&cl1, STARPU_RW, data_A, 0);
starpu_task_insert(&cl2, STARPU_RW, data_B, 0);
```

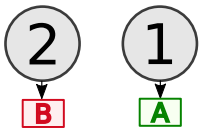
(f) Soumission de la tâche 1 puis la tâche 2 sur deux données A et B.



(g) Graphe produit.

```
starpu_task_insert(&cl2, STARPU_RW, data_B, 0);
starpu_task_insert(&cl1, STARPU_RW, data_A, 0);
```

(h) Soumission de la tâche 2 puis la tâche 1 sur deux données A et B.



(i) Graphe produit.

FIGURE 1.11 : Déduction de dépendances dans le modèle de programmation STF.

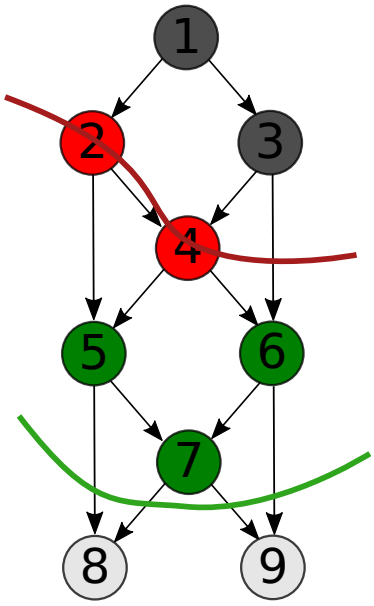


FIGURE 1.12 : Exemple de progression des fronts de soumission et d'exécution. Exemple de progression des fronts de soumission (ligne verte) et d'exécution (ligne rouge). Les tâches gris clair (8 et 9) ne sont pas encore soumises, les tâches vertes (5, 6 et 7) sont soumises et en attente d'exécution, les tâches rouges (2 et 4) sont en cours d'exécution, les tâches gris foncé (1 et 3) sont terminées.

Une fois qu'elles sont toutes deux soumises, l'ordonnanceur peut les faire exécuter en parallèle sur deux ressources différentes puisqu'elles sont toutes deux prêtes. Dès qu'une de ces tâches se terminera, l'ordonnanceur pourra débloquer d'éventuelles tâches dont c'était la dernière dépendance non-validée, et les faire exécuter.

StarPU n'attend pas que l'ensemble des tâches soient soumises avant de commencer à exécuter le graphe de tâches : puisque les tâches sont soumises dans un ordre séquentiel, toute tâche n'ayant pas de dépendance en entrée peut commencer à être exécutée, pendant que le thread principal de l'application soumet la suite du graphe. L'ordre séquentiel garantit en effet qu'il ne pourra pas y avoir de nouvelles dépendances entrantes pour cette tâche, seulement d'éventuelles soumissions de tâches qui dépendront de cette tâche.

Comme illustré à la Figure 1.12, on a ainsi deux *fronts* qui se dégagent : le *front de soumission* correspond à la progression du thread principal de l'application dans sa soumission du graphe de tâches, tandis que le *front d'exécution* correspond à la progression de l'exécution effective des tâches. Chaque tâche du graphe de tâches est ainsi dans l'ordre d'abord non soumise, puis soumise (elle passe le front de soumission), puis en cours d'exécution (elle passe le front d'exécution), puis terminée.

1.3.1.2 Ordre de soumission et chemin critique

Étant donné que les tâches indépendantes peuvent être soumises dans n'importe quel ordre respectif sans changer le graphe produit, plusieurs ordres de soumission des tâches peuvent produire le même résultat, c'est-à-dire que le graphe de tâches inféré par le moteur d'exécution est identique.

Les performances d'exécution obtenues peuvent cependant être différentes selon l'ordre de soumission choisi. Cet ordre va en effet impacter l'ordre d'exécution, et donc les performances obtenues, avec un effet plus ou moins prononcé selon la politique d'ordonnancement. Par exemple, la politique d'ordonnancement *eager* de StarPU, la plus simple, a un comportement FIFO (First-In, First-Out) : quand une ressource de calcul devient disponible, la tâche qui est prête depuis le plus longtemps va être attribuée à cette ressource. Lorsque cette tâche se termine, un ensemble de tâches (dont c'était la dernière dépendance) deviennent prêtes. Ces tâches sont alors débloquées en conservant l'ordre de soumission. Elles sont alors exposées à l'ordonnanceur dans cet ordre, et sont donc ordonnancées dans cet ordre. Cela impacte alors les décisions de placement de ces tâches, et de là les performances obtenues.

Des politiques d'ordonnancement plus évoluées sont fournies par StarPU, qui décident du placement des tâches en prenant en compte la performance de chaque tâche selon le type de ressource de calcul disponible ou la localité des données. Elles apportent des gains de performances notables car les décisions ainsi prises les rapprochent plus ou moins loin de l'ordonnancement optimal du graphe. Un des aspects primordiaux d'un tel ordonnancement évolué est la notion de chemin critique. La terminaison de certaines tâches va en effet débloquer des tâches dont la terminaison va débloquer d'autres tâches, etc., formant ainsi des chaînes de tâches que l'on est obligé d'exécuter en séquences puisque les tâches doivent s'attendre les unes les autres. Les tâches en tête de telles chaînes doivent être exécutées en priorité, pour éviter que de telles chaînes soient exécutées à la traîne, et retardent ainsi la terminaison de l'ensemble du graphe de tâches. Déterminer ces chaînes automatiquement lors de la soumission serait cependant coûteux, alors qu'il est en général relativement simple pour l'application de soumettre les tâches par ordre de priorité, fournissant ainsi déjà à l'ordonnanceur les tâches dans l'ordre dans lequel il est préférable de les exécuter.

Ceci étant, il n'est pas nécessaire d'imposer que l'ordre de soumission suive les priorités de tâches. StarPU permet en effet à l'application d'indiquer la priorité des tâches sous forme numérique, que l'ordonnanceur peut alors utiliser pour trier les tâches prêtes. Il revient alors à l'application de déterminer les priorités à utiliser pour favoriser l'exécution des chemins critiques. Le front d'exécution des tâches aura alors tendance à suivre cet ordre des priorités plutôt que l'ordre induit par le front de soumission. Il reste cependant préférable d'utiliser un ordre de soumission relativement proche de l'ordre

```
...
int vec[k*n] = {...}; // Vecteur de données de calcul
/* On souhaite distribuer ce vecteur sur k nœuds,
en faisant k sous-vecteurs de taille n. */
starpu_data_handle data_h[k];
for(int i=0 ; i<k ; i++) { /*On itère sur le nombre de sous données*/
    starpu_vector_data_register(&data_h[i]  /*Handle pour un sous vecteur de taille n*/,
                                STARPU_MAIN_RAM,
                                &vec[i*n],  /*Pointeur vers données du sous-vecteur*/,
                                n,          /*Taille du sous vecteur*/,
                                sizeof(int) /* Taille d'un élément */);
    starpu_mpi_data_register(   data_h[i],
                                i,          /*Tag utilisé pour les communications*/,
                                i           /*nœud propriétaire de la donnée data_h[i]*/);
}
...
```

Code 1.5 : Exemple de distribution des données sur k nœuds.

des priorités, pour que les fronts de soumission et d'exécution progressent plus ou moins en parallèle, allégeant ainsi le travail de l'ordonnanceur dans la gestion des tâches, en évitant d'accumuler des tâches non prioritaires.

Déterminer des priorités au moins grossières est en général relativement aisé pour le programmeur applicatif. Il est cependant possible de déléguer cela à un compilateur, dont l'analyse polyédrale peut capturer la structure du graphe induit par le code de soumission de tâches, et en déterminer les chemins critiques et de là des priorités à utiliser.

1.3.2 StarPU-MPI, la version distribuée de StarPU

StarPU propose de distribuer le calcul en utilisant MPI [7]. Le code d'application est alors écrit selon le modèle SPMD et l'on peut lancer le code de StarPU de la même manière qu'un code MPI. Un des partis pris de StarPU est que la distribution des tâches aux nœuds n'est pas faite dynamiquement car l'on cherche à éviter que des synchronisations entre les nœuds deviennent nécessaires pour atteindre un consensus sur la question « Quel nœud exécute quelle tâche ? ».

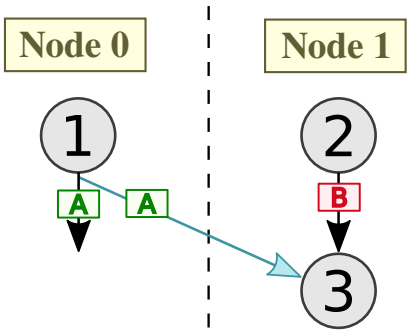
On préfère distribuer statiquement les tâches aux nœuds, en utilisant une politique déterministe qui requiert uniquement des informations que les nœuds possèdent déjà, et donc éviter des message de consensus. En pratique, StarPU propose de distribuer typiquement les données de manière statique, chaque nœud étant le propriétaire unique d'une donnée. Par exemple, le code 1.5 alloue un vecteur composé de k blocs de n valeurs, que l'on distribue sur k nœuds MPI. Le vecteur est d'abord enregistré bloc par bloc auprès de StarPU, produisant pour chaque bloc i un handle `data_h[i]`. Le handle est ensuite enregistré auprès de StarPU-MPI : on précise que pour le bloc i, on

```
starpu_mpi_data_register(data_A, 0, 0);
starpu_mpi_data_register(data_B, 1, 1);

// tâche exécutée par le nœud 0 (où la donnée A réside)
starpu_mpi_task_insert(&cl1, STARPU_W, data_A, 0);

// tâches exécutées par le nœud 1 (où la donnée B réside)
starpu_mpi_task_insert(&cl2, STARPU_W, data_B, 0);
starpu_mpi_task_insert(&cl3, STARPU_R, data_A, STARPU_RW, data_B, 0);
```

(a) Code STF distribué.



(b) Graphe de tâches distribué déduit du code STF 1.13a.

FIGURE 1.13 : Exemple d'exécution de code STF distribué.

utilisera le tag i pour communiquer son contenu (ce fonctionnement est décrit en détails section 1.3.3), et ce bloc est possédé par le nœud i. Le choix du tag et du propriétaire peut être complètement arbitraire, pourvu que tous les nœuds utilisent les mêmes valeurs pour les mêmes handles. Pour les tags il suffit de numéroter les handles sans ambiguïté, par exemple par coordonnées géométriques pour une matrice. Pour les propriétaires de handles, on peut utiliser n'importe quelle fonction de répartition déterministe, la distribution 2D-bloc-cyclique classique [61] par exemple pour les matrices.

Par défaut une tâche est alors exécutée par le nœud qui possède la première donnée indiquée en écriture de la tâche : c'est une solution simpliste mais surtout déterministe et qui ne nécessite pas de synchronisation entre les nœuds. Étant donné que chaque nœud sait précisément où se trouve chaque donnée, le runtime peut alors automatiser le transfert entre les nœuds lorsqu'une tâche nécessite une donnée distante.

La Figure 1.13 détaille un exemple exhibant ce fonctionnement. La donnée A est possédée par le nœud 0 et la donnée B est possédée par le nœud 1. Ainsi, la tâche 1 qui modifie la donnée A est exécutée sur le nœud 0, tandis que les deux tâches 2 et 3 qui modifient la donnée B sont exécutées sur le nœud 1. Le transfert de la donnée A entre les tâches 1 et 3 est alors initié automatiquement. En pratique lorsque le code de soumission du nœud 0 soumet à StarPU-MPI la tâche 3 (qui sera exécutée par le nœud 1), le

runtime note le fait que le nœud 1 a donc besoin de la donnée A (dont il est propriétaire), et crée pour cela sur le nœud 0 une dépendance qui va envoyer la donnée A au nœud 1 avec un `MPI_Send` dès que la tâche 1 est terminée. De son côté le nœud 1, lors de la soumission de la tâche 3, se rend compte qu'il a besoin pour cette tâche de la donnée A possédée par le nœud 0, et va donc créer une réception avec `MPI_Recv`. Une dépendance est alors ajoutée afin de débloquer la tâche 3 lorsque la réception est terminée. Toutes ces étapes de communications sont réalisées automatiquement et en parallèle de l'exécution des autres tâches, apportant un confort d'implémentation au développeur d'application et un fort entrelacement, les communications étant faites de manière complètement asynchrones.

Éviter la duplication des communications

L'avantage d'utiliser un support d'exécution est que StarPU suit les accès subis par les données, et l'on a donc une vision des modifications des données effectuées par les autres nœuds. En effet quand une tâche est soumise, StarPU met à jour les informations concernant la cohérence des données et peut savoir si une donnée locale est modifiée par une autre tâche, mais aussi suivre les modifications des données des autres nœuds. On peut donc s'appuyer sur ces indications pour éviter d'envoyer la même donnée 2 fois au même nœud. Par exemple pour le code 1.14a, on obtient l'exécution illustrée par la Figure 1.14b. Pour les tâches 2 et 3, le nœud 1 a besoin de la même donnée A. Si l'on se limite aux dépendances des tâches présentées précédemment pour déduire le transfert de données, il faudrait envoyer deux fois cette même donnée car l'on ne sait pas si elle a pu être modifiée. Suivre la cohérence de chaque donnée permet de déduire qu'il s'agit de la même donnée et qu'elle n'a pas besoin d'être renvoyée par le nœud 0 pour la tâche 3. De son côté le nœud 1 voit qu'il peut réutiliser cette même donnée car il est assuré qu'elle n'a pas été modifiée. En revanche la donnée A est modifiée par le nœud 0 avec la tâche 4. Elle doit donc être renvoyée au nœud 1 pour satisfaire la dépendance de la tâche 5, ce qu'il est possible de déduire car les deux nœuds ont l'information que A a été modifiée depuis le dernier envoi. Toutes les informations nécessaires à ce fonctionnement sont déduites du code de soumission, donc aucune communication n'est requise entre les noeuds en dehors de ces transferts de données.

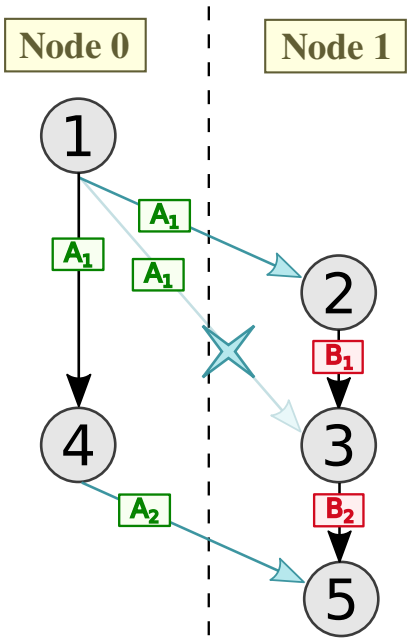
1.3.3 Détails du fonctionnement de StarPU-MPI

La mise en œuvre d'une solution de checkpoint/restart local au sein de StarPU-MPI nécessitera une interaction relativement fine avec le moteur de gestion des communications StarPU-MPI, nous fournissons donc ici les détails d'implémentation qui seront

```
starpu_mpi_task_insert(&cl1, STARPU_W, data_A, 0);
starpu_mpi_task_insert(&cl2, STARPU_R, data_A, STARPU_W, data_B, 0);
starpu_mpi_task_insert(&cl3, STARPU_R, data_A, STARPU_RW, data_B, 0);

starpu_mpi_task_insert(&cl4, STARPU_RW, data_A, 0);
starpu_mpi_task_insert(&cl5, STARPU_R, data_A, STARPU_RW, data_B, 0);
```

(a) Code STF réutilisant la même donnée A sur le nœud 1.



(b) Économie de communications pour le code 1.14a grâce au cache : la donnée A n'a pas besoin d'être retransmise pour la tâche 3.

FIGURE 1.14 : Exemple d'économie de communications grâce au cache.

utiles par la suite.

1.3.3.1 Principe du thread de progression des communications

De nombreuses implémentations du standard MPI ne supportent que le niveau `MPI_THREAD_FUNNELED`, c'est-à-dire que seul le thread ayant appelé `MPI_Init_Thread` peut effectuer des appels aux fonctions MPI. Chaque instance de StarPU utilise donc un *thread de progression* des communications, qui est le seul à utiliser les fonctions MPI. Lorsqu'un autre thread termine par exemple l'exécution d'une tâche dont la donnée doit être envoyée sur un autre nœud, il soumet au thread de progression une requête de communication en l'ajoutant à une liste globale `ready_send_requests`. Cette requête contient toutes les informations utiles pour le transfert de données à effectuer, notamment un tag (qui identifie le handle à envoyer ou recevoir) et les numéros de nœuds source et destination. Le thread de progression traite alors ces requêtes de manière similaire au comportement de MPI : il assure un comportement FIFO entre les requêtes utilisant le même tag. Ainsi, pour un même handle, si le graphe de tâches nécessite que deux versions différentes soient envoyées par MPI, elles seront envoyées (et reçues) dans l'ordre du graphe de tâches, induit par l'ordre de soumission.

De même, les requêtes de réception induites par le graphe de tâches sont soumises au thread de progression en les ajoutant à une liste globale `early_request`.

1.3.3.2 Comportement interne du thread de progression des communications

Pour supporter ce comportement FIFO, StarPU-MPI n'utilise cependant pas directement les tags MPI. En effet, diverses implémentations MPI supportent très mal le cas d'utilisation d'un grand nombre de tags. Le thread de progression de StarPU-MPI réimplémente donc le fonctionnement des tags des handles par-dessus l'implémentation MPI, en utilisant seulement deux tags MPI. Pour éviter la confusion entre les tags utilisés par StarPU pour les handles et les tags utilisés dans les appels MPI, on appellera ces derniers *tags MPI*, le terme *tag* sera utilisé pour désigner les tags associés aux handles.

L'envoi d'un message par StarPU-MPI se fait alors en deux temps. Le thread de progression envoie d'abord sur le tag MPI `MPI_TAG_ENVELOPE` une enveloppe contenant les méta-données (type de message, tag du handle, ...). Il envoie ensuite sur le tag MPI `MPI_TAG_DATA` les données proprement dites. Cette approche garantit que pour une paire de nœuds source et destination donnée, l'ordre d'envoi et de réception des enveloppes correspond strictement à l'ordre d'envoi et de réception des données. Cet ordre correspond à l'ordre des requêtes dans la liste `ready_send_requests`.

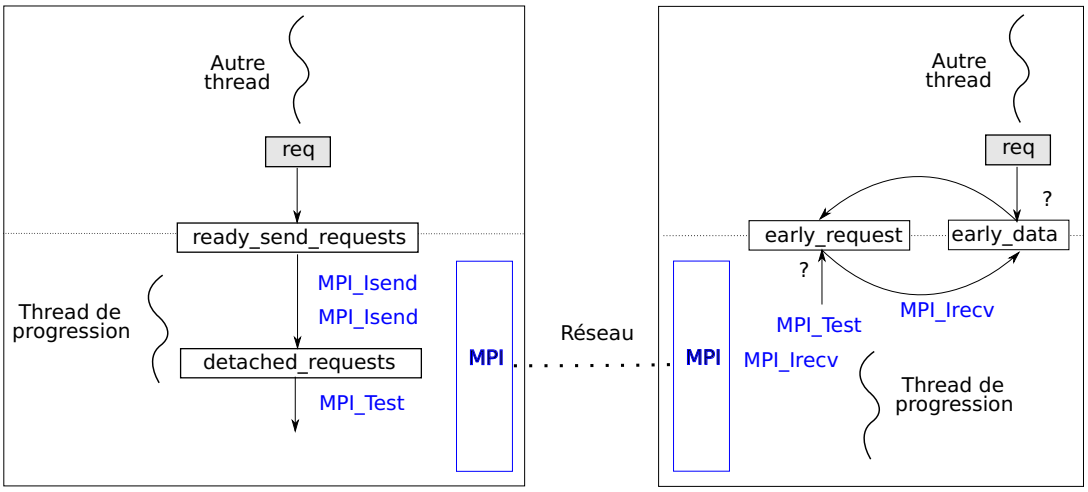


FIGURE 1.15 : Fonctionnement interne du thread de progression de StarPU-MPI. À gauche, le thread de progression soumet les requêtes d'envoi à MPI, d'abord l'enveloppe puis la donnée. À droite, le thread de progression reçoit l'enveloppe, et essaie de trouver une correspondance, pendant que le thread de soumission de requêtes essaie de trouver une correspondance.

Inversement pour recevoir des messages, le thread de progression conserve en permanence une requête de réception d'enveloppe soumise sur le tag MPI `MPI_TAG_ENVELOPE` avec la source `MPI_ANY_SOURCE`. Lorsque MPI indique que cette requête est terminée, le thread peut lire dans l'enveloppe reçue les méta-données du message, faire correspondre ce message avec la requête correspondant dans la liste `early_request`, et poster alors sur le tag MPI `MPI_TAG_DATA` la réception des données associées. L'ordre des requêtes tel que soumis au thread de progression émetteur est ainsi de nouveau respecté.

Ce mode de fonctionnement permettra par ailleurs d'étendre le fonctionnement de StarPU-MPI avec des *messages de service* qui seront utiles pour la gestion de la progression des checkpoints.

Nous décrivons maintenant plus précisément le traitement des requêtes au sein du thread de progression, illustré à la Figure 1.15

Requêtes d'envoi

Les requêtes sont soumises au thread de progression en les plaçant dans la liste `ready_send_requests`. Le thread de progression traite les requêtes de cette liste une par une dans l'ordre. Pour chacune d'entre elles, il soumet l'envoi de l'enveloppe puis l'envoi des données à MPI. La requête est alors placée en attente dans la liste `detached_requests`. Le thread teste périodiquement la terminaison des requêtes recensées par cette liste. Lorsque MPI indique qu'une requête est achevée, le thread retire la requête de la liste,

et appelle une éventuelle fonction de callback renseignée dans la requête.

Requêtes de réception

L'ordre de réception des messages provenant de différents nœuds MPI ne correspond en général pas exactement à l'ordre dans lequel les réceptions ont été soumises d'après le graphe de tâches. En effet, l'ordre dépend potentiellement du temps dont les autres nœuds ont besoin pour calculer les données. Le thread de progression doit donc correctement distribuer les messages aux différentes requêtes de réception qui lui sont soumises, en utilisant le tag et la source indiqués dans l'enveloppe du message.

Les requêtes de réception sont normalement soumises au thread de progression en les plaçant dans la liste `early_request`. Lorsque le thread reçoit une enveloppe, il cherche dans cette liste une requête correspondant. Si elle existe, il peut poster immédiatement la réception dans le handle indiqué par la requête. S'il n'y a pas encore de requête correspondant dans la liste `early_request`, cela signifie que la soumission du graphe de tâches n'a pas encore assez progressé, ou alors que des tâches utilisent encore le handle qui sera utilisé pour la réception. Le thread crée alors un handle temporaire, poste la réception des données dans ce handle, et crée une requête temporaire en utilisant les informations de l'enveloppe. Il place cette requête dans une liste `early_data` des données reçues en avance.

Inversement, quand une requête de réception est soumise au thread de progression, avant de la placer dans la liste `early_request` comme indiqué plus haut, on vérifie s'il y a une requête temporaire correspondant dans la liste `early_data`. Dans ce cas, les données sont copiées depuis le handle temporaire vers le handle de la vraie requête, et le traitement est terminé.

Les vérifications dans les listes `early_request` et `early_data` sont effectuées en suivant l'ordre de ces listes, pour utiliser la première occurrence d'une requête correspondant, ce qui permet de respecter l'ordre FIFO.

1.3.3.3 Support des priorités de messages

Dans les sections précédentes, nous avons souligné que l'ordre FIFO était toujours respecté, ce qui permet aux messages d'être *in fine* reçus dans l'ordre de soumission. Nous avons cependant vu à la section 1.3.1.2 qu'il était utile de pouvoir définir des priorités, qui guident l'ordonnanceur pour éventuellement exécuter les tâches dans un ordre différent de l'ordre de soumission. De même, il est utile de pouvoir attribuer des priorités aux communications MPI. La liste `ready_send_requests` est donc en fait triée par priorité des requêtes, ce qui contredit justement le respect de l'ordre FIFO de soumission.

Il se trouve que StarPU ne place jamais dans la liste `ready_send_requests` deux requêtes pour deux versions différentes d'un même handle. Au plus, il place plusieurs requêtes pour une même version envoyée à différents nœuds destination. En effet, lorsqu'une tâche est soumise pour modifier un handle, elle est mise en attente de toute tâche ou envoi MPI qui nécessite la version précédente du handle. Pour éviter d'avoir un usage mémoire incontrôlé, StarPU ne prend pas de lui-même l'initiative d'effectuer une copie de la version précédente qui permettrait d'exécuter en concurrence l'envoi de l'ancienne version et la fabrication de la nouvelle version.

Ainsi, même si les requêtes de la liste `ready_send_requests` sont réordonnées par priorités, elles sont toujours *causalement indépendantes*, le changement d'ordre n'a pas d'impact sur le respect de la causalité dans la sémantique séquentielle du modèle de programmation STF.

Cependant, lorsque nous souhaiterons mettre en place le rejeu d'un log de message (à la section 3.2.2), nous soumettrons éventuellement au thread de progression des requêtes pour de multiples versions d'un même handle. Il sera alors nécesaire, lors du tri par priorité, de ne pas intervertir des messages pour un même handle, pour respecter l'ordre de causalité.

1.3.4 Bilan

StarPU propose de décrire le workflow d'une application sous forme de graphe de tâches, en insérant les tâches de manière séquentielle selon le modèle STF. En parallèle de la soumission du graphe de tâches, un ordonnanceur va envoyer les tâches sur les ressources disponibles en respectant les dépendances. L'ordonnanceur peut être choisi parmi différentes politiques qui permettent d'être en adéquation avec la structure du graphe de tâches et les ressources disponibles. La complexité pour atteindre de bonnes performances est de faire en sorte que des politiques d'ordonnancement relativement simples puissent suivre au mieux le chemin critique de l'application afin de se rapprocher de l'exécution optimale. Par exemple si l'on a des ressources de calcul hétérogènes il est intéressant d'utiliser un ordonnanceur qui prend en compte les modèles de performance des tâches selon les différentes ressources, alors que l'intérêt de telles politiques est limité s'il l'on utilise uniquement des ressources homogènes comme des cœurs CPU.

L'ordre dans lequel est écrit l'application est aussi déterminant car la plupart des ordonnanceurs priorisent implicitement les tâches insérées en amont, mais il est possible de moduler ce comportement en utilisant un ordonnanceur prenant en compte des priorités que le développeur a explicitement affectées aux tâches. Néanmoins un ordonnanceur nécessite des ressources de calculs de manière plus ou moins importante selon la quantité de paramètres qu'il prend en compte, et l'on doit avoir des tâches suffisamment longues pour que ce surcoût soit négligeable. En mode distribué StarPU

utilise une distribution statique des données et l'on n'a pas d'ordonnancement dynamique entre les nœuds. En revanche StarPU intègre une couche de communication utilisant MPI qui permet d'automatiser les transferts selon les dépendances entre les nœuds.

Du point de vue des possibilités pour la tolérance aux pannes, StarPU est intéressant grâce à plusieurs points : son modèle STF, le fait que chaque nœud puisse suivre de manière locale l'évolution des données sur l'ensemble des nœuds, et le fait d'embarquer sa propre couche de communication. Ce sont ces trois points sur lesquels reposent nos contributions et il devrait être possible d'implémenter les solutions présentées dans n'importe quel support d'exécution présentant ces trois propriétés.

Chapitre 2

Checkpoint de niveau application pour un runtime avec le modèle STF

Dans cette partie nous proposons une solution permettant la sauvegarde des données de calcul afin de reprendre l'exécution après une panne. Nous commençons par nous demander s'il est intéressant de s'appuyer sur les solutions de tolérance aux pannes existantes afin de répondre à la problématique des sauvegardes pour le cas de StarPU. Puis nous verrons que certaines spécificités de notre support d'exécution permettent une autre approche, amenant à une solution moins coûteuse en taille de sauvegarde. La combinaison du modèle de programmation STF (Section 1.3.1) et de la sémantique SPMD proposée par StarPU nous permet de définir simplement des checkpoints cohérents (1.2.2.2), en insérant des requêtes de sauvegardes de manière statique dans le code d'application. Cette approche permet de décrire sans ambiguïté le contenu des checkpoints, qui est alors déterministe et est connu avant même la compilation.

Le déterminisme du contenu des checkpoints est une propriété clé qui nous permettra d'anticiper les sauvegardes. Notre méthode permet d'effectuer des sauvegardes de manière incrémentale, plutôt qu'arrivant par vague sur le support mémoire comme l'on trouve généralement dans l'état de l'art. Notre approche est cependant différente de l'approche incrémentale habituelle, grâce à l'asynchronisme de StarPU : il est possible de commencer à sauver les données dès qu'elles sont disponibles, sans qu'il ne soit pour autant nécessaire d'attendre que tout le calcul atteigne l'état décrit par le checkpoint. Par conséquent nous nous distinguons des solutions qui se focalisent sur la capacité d'effectuer une sauvegarde en interrompant l'exécution de l'application, car nous pouvons anticiper le contenu des checkpoints avant que les données ne soient effectivement créées. En contrepartie, les checkpoints doivent être soumis statiquement dans le code d'application, avec une

interface qui est très proche de ce que l'on peut trouver dans les solutions de checkpoints de niveau application comme FTI ou VeloC. Savoir où insérer les checkpoints dans le code d'application n'est pas forcément critique, car un mauvais placement n'empêchera pas le fait de pouvoir redémarrer. En revanche la stratégie de placement des checkpoints aura un impact sur le coût des sauvegardes et la quantité de travail perdu en cas de panne.

Comme StarPU suit les modifications de données, on peut savoir si une donnée a vraiment besoin d'être sauvée lorsqu'une sauvegarde est requise. En effet il n'est pas nécessaire de sauver une donnée si elle n'a pas été modifiée depuis le dernier checkpoint ou depuis l'état initial, ce que le runtime sait déterminer. On peut donc sauver des checkpoints de manière différentielle, sans devoir pour autant utiliser des ressources pour comparer les données.

De plus, en choisissant de sauvegarder les checkpoints sur les autres nœuds de calcul, on pourra bénéficier de la réplication de données déjà effectuée pour les besoins du calcul. StarPU suit déjà les données, leurs modifications et leurs mouvements, et il est possible d'utiliser ces informations afin d'éviter d'envoyer une donnée deux fois si un nœud a besoin de cette donnée pour le calcul et s'il doit également sauver cette donnée pour le checkpoint d'un autre nœud. Les expériences réalisées permettent de valider notre approche, mais aussi d'étudier quelles sont les propriétés qui permettraient à une application de bénéficier au mieux de la stratégie de sauvegarde sur les nœuds. En effet certaines classes auront du mal à bénéficier de la sauvegarde sur les nœuds de calcul, mais dans ces cas, la littérature est suffisamment fournie pour apporter une solution qui reste compatible avec nos checkpoints. Sauvegarder les checkpoints sur les autres nœuds de calcul est en revanche beaucoup moins approfondi dans la littérature, et l'état de l'art ne va pas aussi loin que ce que nous pouvons réaliser en faisant interagir la tolérance aux pannes avec le runtime.

2.1 Quelle approche adopter pour réaliser des checkpoints avec StarPU ?

On peut commencer par supposer qu'il est nécessaire de considérer l'état interne de StarPU comme étant une partie importante de l'état de l'application, et que l'on doive donc le sauvegarder à chaque checkpoint. Dans ce cas on peut utiliser une solution de checkpoint au niveau système afin de sauver l'ensemble du contexte de l'application et du runtime, selon une des solutions évoquées en Section 1.2.2.1. De plus le comportement de StarPU n'est pas déterministe du point de vue du système, car les tâches ne sont pas toujours exécutées dans le même ordre et les transferts de données non plus. Il faudrait donc utiliser des algorithmes complexes pour s'assurer que l'on crée des checkpoints

cohérents. Également cette approche impose d'interrompre le calcul, au niveau d'un seul nœud si l'on effectue des checkpoint locaux non coordonnés, ou bien interrompre l'ensemble des nœuds dans le cadre du checkpoint coordonné. Aller dans cette direction n'est pas intéressant car l'application et son runtime sont alors considérés comme n'importe quelle autre application et l'intérêt de recherche est inexistant dans le cadre d'un runtime comme StarPU, car les propriétés d'un tel support d'exécution ne sont alors pas exploitées.

Si l'on décide d'effectuer des checkpoints de niveau application (voir section 1.2.2.1) à l'intérieur de StarPU, il faudrait instrumenter tout le code et insérer les checkpoints dans l'ensemble des threads ce qui n'est pas raisonnable. Implémenter cela serait très complexe, voire impossible, et les bénéfices ne seraient pas supérieurs à ceux que l'on propose par la suite.

Néanmoins le problème se simplifie si l'on prend le temps d'étudier le fonctionnement de StarPU. La force d'un runtime à base de tâches vient du fait que la soumission effectuée par le code de l'application et l'exécution du calcul sont distincts. On avait vu en Section 1.3.1 que le code de l'application soumet des tâches au support d'exécution, ce dernier s'occupant d'exécuter les tâches en respectant les dépendances. Ainsi l'application pilote le runtime, et l'état interne de ce dernier n'est en réalité pas critique et n'a donc pas besoin d'être sauvegardé. En effet StarPU ne garde pas d'informations quant aux tâches terminées, et donc le passé révolu n'a pas d'incidence sur l'état interne d'un runtime. Seules les tâches soumises et celles en cours d'exécution ont un impact sur l'état interne. Le runtime s'occupe simplement d'exécuter des tâches lorsque les données nécessaires sont prêtes ; il n'est pas nécessaire d'avoir connaissance des tâches qui ont eu lieu pour fonctionner ainsi. Pour reprendre un calcul, il suffit juste d'être en capacité de fournir au runtime l'ensemble des tâches qu'il reste à exécuter ainsi que les données initiales nécessaires.

Le runtime StarPU n'ayant pas un état interne critique qui doit être conservé et rétabli dans le cas où une panne se produit, on peut se concentrer uniquement sur les données du code d'application. On pourrait laisser StarPU gérer les checkpoints dynamiquement au cours de l'exécution, mais il serait très compliqué de commencer les travaux par là. Aussi on préfère d'abord insérer les checkpoints statiquement dans le code d'application. Outre le fait d'être plus facile à implémenter que des checkpoints dynamiques, on verra qu'utiliser des checkpoints statiques permet de bénéficier au mieux des propriétés de StarPU. Également la reprise après panne peut être réalisée plus simplement qu'avec une approche dynamique, étant donné que le code d'application contient les marques des sauvegardes.

Nous partons donc du principe que les checkpoints sont insérés statiquement dans le code d'application pour la suite de nos contributions. Cette approche nous permet

notamment de séparer sans ambiguïté le graphe des tâches soumises du graphe des tâches qu'il reste à soumettre, et ce de manière uniforme pour tous les nœuds sans aucune synchronisation explicite.

Autres travaux de tolérance aux pannes pour les supports d'exécution distribués à base de tâches

On peut noter que l'on ne peut pas s'inspirer des travaux réalisés pour Parsec [20], qui est un autre runtime à base de tâches. En effet ces travaux reposent sur le fait que Parsec peut récupérer à volonté des informations sur le graphe de tâches, car il utilise une représentation *PTG* (Graphe de Tâches Paramétrique) au cœur même de son fonctionnement. StarPU n'a pas la même approche fondamentale d'aborder l'exécution d'une application sous forme de graphe de tâches, et ne peut pas récupérer des informations sur les tâches terminées ou les tâches qui ne sont pas encore soumises.

Les travaux réalisés pour OmpSs [60] se focalisent sur la capacité de ré-exécuter une tâche qui a échoué. Si les travaux sont intéressants pour nous à première vue, en réalité on ne peut pas s'en inspirer ; en effet l'approche proposée y repose sur le fait que les tâches intègrent elles-mêmes les appels MPI. Dans StarPU, les appels MPI sont explicitement déduits du graphe de tâches et ne sont donc pas contenus dans le code exécuté par une tâche. La ré-exécution de tâches n'est donc pas complexe dans notre cas, étant donné que les données lues par la tâche sont toutes présentes en local au moment où elle se lance. Le résultat de l'exécution d'une tâche est donc toujours déterministe car il ne peut pas être influencé par des appels MPI au cours de l'exécution de la tâche. On notera en particulier que ces travaux se focalisent sur la ré-exécution de tâche lorsqu'une erreur se produit pendant son exécution, mais cette solution n'est pas applicable dans le cas d'une panne franche d'un nœud. Le contexte de cette thèse porte précisément sur ces pannes franches, la ré-exécution simple de tâche ne permet donc pas de répondre à l'objectif de cette thèse.

Le support d'exécution Spark propose une approche différente [56]. Ces travaux sont intéressants car permettent d'obtenir de bons résultats, mais ne sont une nouvelle fois pas applicables ici. En effet le principe repose sur le fait que les tâches sont créées par l'exécution de tâches précédentes. Ainsi le point de reprise peut être facilement défini et représenté de manière légère. Dans StarPU le flux de tâches vient du code de soumission : chaque exécution de tâche débloque des dépendances qui permettent à de nouvelles tâches de se lancer. A contrario dans Spark, de nouvelles tâches sont créées par une tâche qui se termine et cette différence est majeure dans l'approche qu'il conviendrait d'adopter pour implémenter un protocole de tolérance aux pannes.

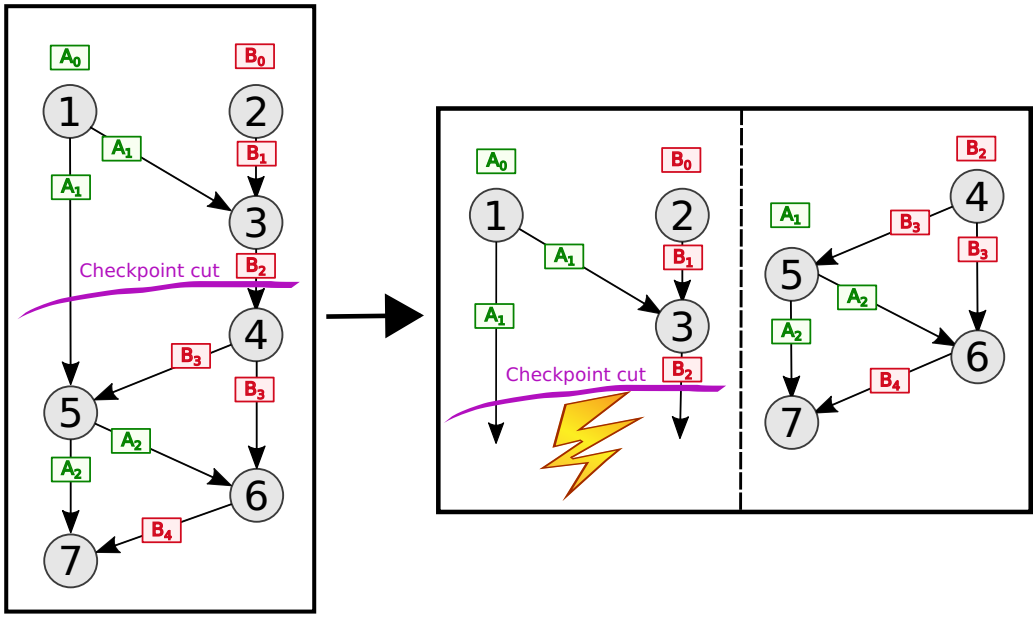


FIGURE 2.1 : Équivalence des graphes : Exécuter le graphe de gauche est équivalent à équivalent à exécuter successivement les graphes de droite.

2.1.1 Un checkpoint cohérent, asynchrone, non-coordonné, incrémental et différentiel

Cette partie explique comment nous définissons un checkpoint dans le cadre de StarPU. On distingue les propriétés acquises grâce au modèle de programmation STF, et celles acquises par des spécificités de StarPU. On pourra réutiliser tout ce qui est défini ici pour étendre les modèle de sauvegarde des checkpoints.

2.1.1.1 Comment des checkpoints s'insèrent-ils au sein du modèle de programmation STF ?

Il est utile de discuter d'abord du principe même de checkpoint dans le cadre du modèle de programmation STF et SPMD. En effet, celui-ci nous apporte des informations précieuses sur la structure du calcul, il s'agit ici d'en profiter du mieux possible pour optimiser finement la prise de checkpoints.

Comment séparer la soumission en deux ?

L'avantage du modèle STF est qu'il permet de s'affranchir du raisonnement sur le parallélisme. Ainsi pour faciliter la compréhension on peut considérer pour l'instant que les tâches sont exécutées sans parallélisme. Une tâche soumise par l'application est directement exécutée, et la fonction `task_insert` retourne dès que l'exécution de la

tâche est terminée. Bien que StarPU fonctionne différemment, raisonner ainsi ne remet pas en question ce qui va être présenté tout en facilitant grandement la compréhension.

Avec le modèle STF il est facile de séparer un graphe en deux parties : on peut séparer deux portions du code de l'application pour obtenir deux sous-graphes, les données finales du premier graphe étant les données initiales du second, comme on peut voir sur la Figure 2.1. Ainsi, insérer un checkpoint à un endroit de la soumission permet de séparer l'exécution en deux graphes de tâches, le premier graphe étant le graphe des tâches terminées et le second celui des tâches qu'il reste à soumettre. On peut donc sauvegarder les données finales du premier graphe en les plaçant dans un checkpoint. Si l'application est interrompue, on pourra alors initialiser les données sur la base du checkpoint. Soumettre les tâches qu'il reste à effectuer est simple, il suffit de reprendre la soumission à l'endroit où le checkpoint a été inséré.

Sur la Figure 2.1, on voit que si une panne se produit après le checkpoint, il suffit d'initialiser les données A et B à leurs valeurs respectives A_1 et B_2 et de reprendre la soumission du graphe à partir de la tâche 4. Si toutes ces informations sont sauvées dans le checkpoint, on sait qu'on pourra redémarrer. Exprimer les données à sauvegarder est facile, mais déterminer le point de reprise dans l'application est plus compliqué. On sait qu'il faut reprendre à la tâche juste après la soumission de checkpoint dont sont issues les données, mais atteindre cette instruction en ne faisant rien avant peut être compliqué selon la structure du code d'application.

Et en distribué ?

On a vu qu'insérer un checkpoint dans le code de l'application permet de distinguer sans ambiguïté les tâches qui ont été soumises ou non. Mais dans le domaine du calcul distribué ceci n'est pas suffisant, car il faut aussi se préoccuper de la cohérence. L'avantage d'avoir un flux de tâche séquentiel est que l'on a intrinsèquement une vraie propriété de causalité entre les tâches : une tâche ne peut pas dépendre d'une tâche qui sera soumise après celle-ci. Même s'il est SPMD, un code MPI ne bénéficie pas de cette même propriété car l'information de ce que font les autres nœuds est masquée.

On avait pu voir que dans un code MPI, un `MPI_Irecv` peut être écrit dans le code source avant le `MPI_Send` correspondant (Section 1.3). On avait en particulier remarqué qu'insérer un checkpoint entre ces deux appels créait une incohérence. Dans un code utilisant le modèle STF et particulièrement StarPU-MPI qui gère les communications lui-même, ce cas ne peut pas se présenter. Chaque nœud soumet le même graphe de tâches et insérer un checkpoint statiquement entre les tâches permet de créer un checkpoint global qui sépare les tâches soumises des tâches qu'il reste à soumettre. Pour que la séparation soit la même sur tous les nœuds, nous devons nous assurer que le checkpoint soit inséré au même endroit dans la soumission pour l'ensemble des

```
...
int A=0, B=0;

starpu_data_register(&A, Ah);
starpu_data_register(&B, Bh);

// Node 0 owns data A
starpu_mpi_data_register(A_h, 0);
// Node 1 owns data B
starpu_mpi_data_register(B_h, 1);

// Task 1 reads and writes data A
// and is executed on node 0
task_insert(&task1, A_h, RW);

checkpoint();

// Task2 reads and writes data B
// and reads data B
// and is executed on node 1
task_insert(&task2, B_h, RW, A_h, R);
```

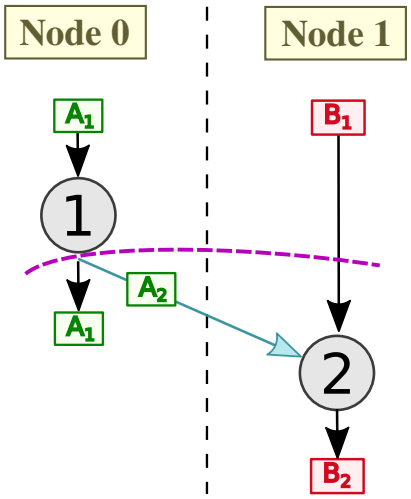


FIGURE 2.2 : Exemple d'un checkpoint placé statiquement dans un code STF. Il n'y a pas d'ambiguïté que la nouvelle valeur de A est dans le checkpoint, tandis que c'est la valeur initiale de B qui est dans le checkpoint.

nœuds, ce qui est effectué intrinsèquement par l'insertion statique des checkpoints dans le code d'application. Les checkpoints sont donc implicitement coordonnés par le code de l'application sans qu'une synchronisation ait lieu en pratique, ce qui ne crée pas de blocage entre les nœuds qui aurait pour effet de ralentir la progression ; cette approche n'ajoute donc aucun overhead important en comparaison d'une exécution de StarPU sans checkpoint.

Afin de faciliter la compréhension, nous pouvons là aussi adopter le raisonnement simplifié vu précédemment, dans lequel chaque tâche est exécutée séquentiellement dès sa soumission. On considère alors que l'on a des coordinations entre les nœuds, une tâche restant bloquée sur tous les nœuds tant que celui qui est censé l'exécuter ne l'a pas terminée.

On a pu voir en 1.3.2 que les transferts de données sont automatiquement gérés par le runtime et sont initiés par l'insertion d'une tâche. En effet ce n'est que lorsqu'une tâche est insérée que le runtime déduit qu'un nœud doit envoyer une donnée et qu'un autre nœud doit la recevoir. Il est important de noter que du point de vue de la soumission, les transferts nécessaires pour exécuter une tâche et l'exécution de cette même tâche sont atomiques ; une unique ligne dans le code d'application permet de signifier ces opérations. Les envois et réceptions correspondant sont donc aussi atomiques du point de vue soumission, et il est donc impossible d'insérer un checkpoint entre un envoi et la réception correspondante avec ce modèle. Par exemple sur la Figure 2.2, le message A_2

est initié par la tâche $task_2$, que ce soit du point de vue du nœud 0 ou du nœud 1. Étant donné que le checkpoint est inséré avant la tâche $task_2$, la communication est causalement postérieure au checkpoint pour les deux nœuds. Comme vu en Section 1.2.2.3, ceci nous garantit qu'il est impossible d'avoir des messages orphelins, ce qui nous assure d'avoir un checkpoint global cohérent et que nous ne sommes pas sous la menace d'un effet domino. De la même manière il ne peut pas y avoir de message en transit ce qui permet de nous affranchir de considérer les ré-émissions potentiellement nécessaires. Insérer les checkpoints dans la soumission permet de bénéficier des avantages du STF, à savoir la causalité et le déterminisme. Ces deux propriétés nous immunisent contre l'apparition de checkpoints incohérents. On peut comprendre dès lors qu'il est préférable de placer un checkpoint statiquement dans un flux de tâches déterministe (qui est indépendant de son exécution réelle qui n'est pas déterministe), plutôt que de placer un checkpoint dynamiquement dans un flux d'exécution non déterministe.

Ceci peut être vu sous un autre angle. En insérant un checkpoint statiquement dans un flux de tâches causal et déterministe, on impose le contenu du checkpoint sans ambiguïté. On remarque sur la Figure 2.2 que l'on est assuré que le checkpoint contient la donnée A telle qu'elle a été modifiée par la tâche $task_1$ tandis que la donnée B vaut toujours 0 car n'a pas encore été modifiée par la tâche $task_2$. Si l'on suit ce modèle, dès que l'on souhaite redémarrer on peut restituer sur les nœuds les données A et B telles que sauvegardées lors du checkpoint, et reprendre l'exécution à l'instruction qui suit le checkpoint correspondant. Cette remarque est intéressante car l'on se rend compte que la cohérence du checkpoint est définie sans ambiguïté par les données du checkpoint et le point de reprise correspondant : si l'on sauvegarde ces données, on est assuré d'avoir un checkpoint cohérent.

La ressemblance avec des bibliothèques de checkpoint comme FTI ou VeloC 1.2.2.3 se remarque particulièrement. On pourrait en effet les utiliser en plaçant les checkpoints de la même manière que ce qui a été présenté. Néanmoins, il faudrait créer des points de synchronisation locaux avant les checkpoints, en utilisant la fonction `starpu_task_wait_for_all()` pour attendre que toutes les tâches soumises soient terminées et ainsi obtenir un état cohérent avant de commencer à sauvegarder les données. Ceci limite le parallélisme atteignable en pratique par StarPU et dégrade fortement les performances, sans compter le coût des checkpoints. En pratique, le contenu des checkpoints est le même pour les deux approches, mais on préférera réaliser les sauvegardes en interne du runtime car cela est beaucoup plus efficace. Là où le rôle du support d'exécution est déterminant, c'est que l'on sera capable de constituer ce checkpoint au fur et à mesure de l'exécution, en le composant partiellement dès qu'une des données est disponible, en effectuant un checkpoint incrémental (Section 1.2.2.5).

On peut dès lors quitter le raisonnement simplifié où l'on considérait que chaque

tâche est exécutée séquentiellement.

2.1.1.2 Quels bénéfices sont apportés par StarPU ?

Le runtime StarPU en lui-même apporte des propriétés que l'on peut exploiter pour optimiser notre approche. Les propriétés précédentes étaient spécifiques au modèle de programmation STF/SPMD, qui est utilisé par StarPU.

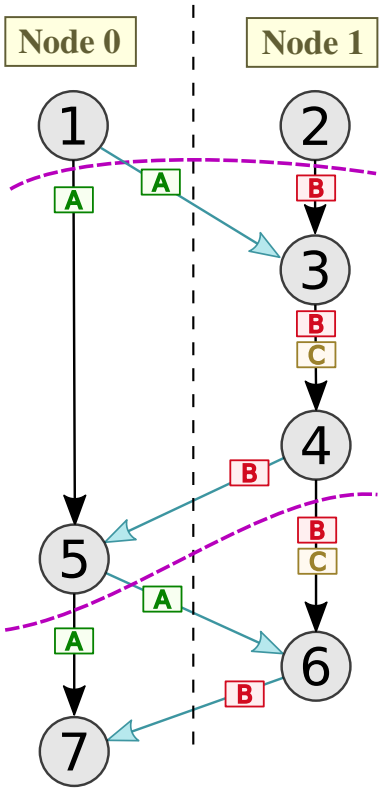
Checkpoint incrémental

StarPU s'occupe d'exécuter de manière asynchrone les tâches, en exécutant une tâche dès que les données nécessaires sont prêtes, ce qui va créer de nouvelles données qui vont débloquer d'autres tâches, tout en exécutant les tâches indépendantes en parallèle. Ainsi, insérer un checkpoint dans le graphe de tâches peut être vu comme insérer une tâche qui va avoir des dépendances en lecture sur les données qui sont identifiées sans ambiguïté comme faisant partie du checkpoint. Le checkpoint peut, de cette manière, être réalisé de manière complètement asynchrone, en parallèle de l'exécution (en réalité le checkpoint n'est pas effectué comme une tâche dans StarPU pour de nombreuses raisons techniques que nous passerons sous silence). Dans le cas de StarPU, il est possible de créer une dépendance sur une donnée et de compléter le checkpoint de manière incrémentale, ce qui a l'avantage d'étaler l'enregistrement du checkpoint dans le temps. La sauvegarde d'une donnée débute donc dès que les tâches précédant l'insertion du checkpoint relâchent l'accès en écriture. On accède alors en lecture sur la donnée ce qui garantit que cette donnée ne sera pas modifiée par les tâches causalement postérieures au checkpoint tant que la sauvegarde n'est pas achevée. Cela est illustré à la Figure 2.3. La sauvegarde créant un accès en lecture sur la donnée, les tâches postérieures qui requièrent un accès en écriture sur cette donnée ne pourront être exécutées tant que la sauvegarde n'est pas terminée, ce cas est appellé *write after read*. A contrario, les tâches prêtes demandant la lecture de cette donnée ne sont pas bloquées par la sauvegarde et peuvent être exécutées en parallèle.

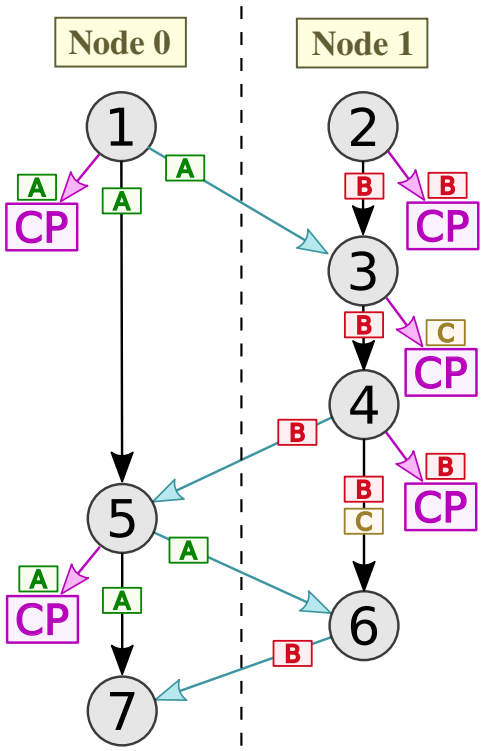
Ainsi avec un support d'exécution basé sur le modèle STF, il est possible d'utiliser des checkpoints statiques pour séparer sans ambiguïté les tâches soumises des tâches qu'il reste à soumettre. Si le runtime gère de lui-même les communications nécessaires au calcul distribué, comme c'est le cas avec StarPU, on a l'assurance que les checkpoints soient cohérents, et qu'aucun message n'est perdu. De plus l'exécution des tâches étant asynchrone, les checkpoints peuvent être réalisés en parallèle du calcul, ce qu'aucune autre solution de checkpoint ne peut faire. Finalement, l'état du checkpoint sauvegardé correspond bien à l'état des données tel que déterminé par le graphe de tâches, et le

```
task_insert(&f1, A, W);
task_insert(&f2, B, W);
checkpoint();
task_insert(&f3, B, RW, A, R, C, W);
task_insert(&f4, B, RW);
task_insert(&f5, A, RW, B, R);
checkpoint();
task_insert(&f6, B, RW, A, R, C, R);
task_insert(&f7, A, RW, B, R);
```

(a) Code source de l'exemple.



(b) Graphe de tâches obtenu avec le code 2.3a.



(c) Sauvegarde effectivement initiés pour le graphe de la Figure 2.3b

FIGURE 2.3 : Illustration du checkpoint incrémental. On remarque par exemple sur la Figure 2.3c que la donnée C est sauvegardée pour le deuxième checkpoint dès que la tâche 3, qui la produit, est terminée. La donnée B, par contre n'est sauvegardée pour le deuxième checkpoint qu'une fois que la tâche 4 (la dernière tâche qui modifie B avant le checkpoint) est terminée. Les transferts de C et de B sont ainsi étalés entre les deux checkpoints.

checkpoint n'a pas besoin de représenter un état par lequel est passé l'application.

Checkpoint différentiel

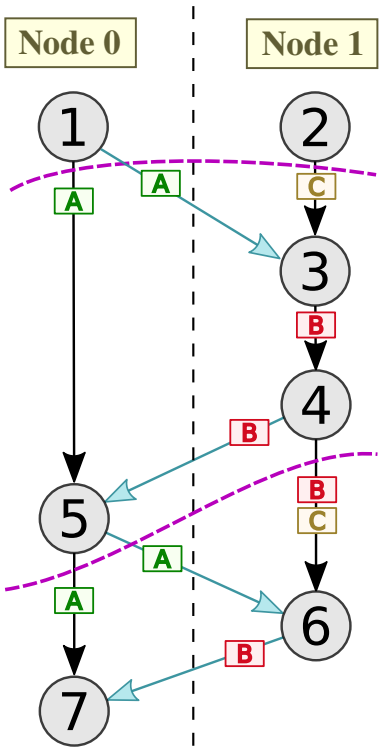
Étant donné que le runtime suit les modifications de données, il est en capacité de déduire l'évolution d'une donnée entre deux checkpoints. Ainsi si cette donnée n'a pas changé entre deux checkpoints, et si cette donnée est stockée sur le même nœud pour les deux checkpoints, il est possible d'éviter une duplication de la communication mais aussi éviter que la donnée soit stockée deux fois. Cela est illustré Figure 2.4. Le mécanisme de checkpoint reprend alors la propriété du checkpoint incrémental, qui permet d'éviter de stocker des données redondantes entre deux checkpoints et ainsi réduire l'empreinte mémoire des checkpoints. Cela crée néanmoins des relations entre les checkpoints, dont il faut tenir compte dans le mécanisme d'éviction de sauvegardes obsolètes.

2.1.1.3 Limites

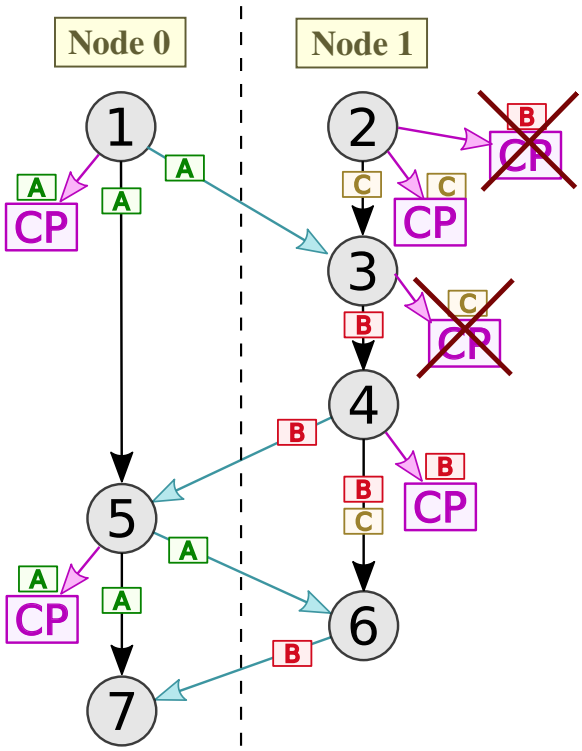
Outre le fait de nécessiter des modifications dans le code d'application, le principal reproche que l'on peut faire à notre approche est que l'on ne sauvegarde qu'un sous-ensemble des contributions. Lorsque le checkpoint est terminé, le runtime a très probablement terminé des tâches dont le résultat n'est pas inclus dans le checkpoint. Il s'agit en réalité d'une conséquence directe du fait que l'on ne bloque pas l'exécution pendant la sauvegarde. Ceci crée donc une tendance naturelle à retarder les checkpoints, et ce retard est directement lié à l'ordre de soumission des tâches et à l'ordonnanceur choisi. Comme on l'avait vu en Section 1.3.1.2, ces deux paramètres sont déterminants pour faire en sorte que le flux d'exécution des tâches suive au mieux le chemin critique. L'ordre de soumission est important étant donné que les ordonnanceurs ont tendance à effectuer les tâches soumises en premier lorsqu'elles sont prêtes. On avait remarqué que dans ces conditions, il est préférable de soumettre les tâches dans un ordre qui correspond à leur exécution afin de coller au mieux au chemin critique. Ceci est d'autant plus important ici car adopter cette philosophie permet aux checkpoints de ne pas être retardés de manière pénalisante. En particulier dans le cas où une tâche est insérée en début de soumission alors qu'elle est exécutée en pratique à la fin du programme à cause d'un concours de circonstance ; cela empêchera la terminaison des checkpoints car ils ont besoin de la donnée de cette tâche en retard pour être complétés. Cette situation est illustrée sur la Figure 2.5.

```
task_insert(&f1, A, W);
task_insert(&f2, B, W, C, W);
checkpoint();
task_insert(&f3, B, RW, A, R, C, R);
task_insert(&f4, B, RW);
task_insert(&f5, A, RW, B, R);
checkpoint();
task_insert(&f6, B, RW, A, R, C, R);
task_insert(&f7, A, RW, B, R);
```

(a) Code source de l'exemple. La différence avec le code 2.3a est que c'est la tâche 2 qui produit la donnée C, au lieu de la tâche 3, qui ne fait que la lire.



(b) Graphe de tâches obtenu avec le code 2.4a.



(c) Transferts effectivement initiés pour le graphe de la Figure 2.4b

FIGURE 2.4 : Illustration de données redondantes. La donnée B n'est pas modifiée par la tâche 2, elle est encore à l'état initial, il n'y a donc pas besoin de la sauvegarder pour le premier checkpoint. De même, entre les deux checkpoints, la tâche 3 lit la donnée C (produite par la tâche 2) mais ne la modifie pas. Il n'y a donc pas besoin de sauvegarder C pour le deuxième checkpoint.

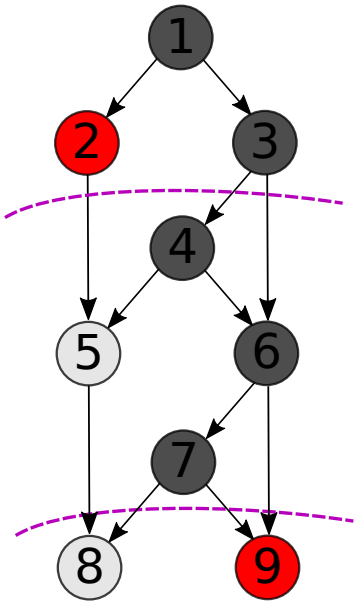


FIGURE 2.5 : Exemple de situation défavorable pour la progression des checkpoints. Les tâches 1, 3, 4, 6 et 7 sont achevées, mais l'exécution de la tâche 2 a été retardée, par exemple parce que l'ordonnanceur a remarqué que seules les tâches 5 et 8 ont besoin de son résultat. Le premier checkpoint ne peut donc pas être achevé, alors même que le deuxième checkpoint est déjà presque complet.

2.1.1.4 Comment bénéficier des mêmes propriétés avec du checkpoint dynamique ?

Les checkpoints que l'on vient de présenter ont ces propriétés parce qu'ils sont implicitement coordonnés par leur insertion statique dans le code. Pour être dynamiques et coordonnés, leur coordination aurait l'obligation d'être explicite. Pour cela les nœuds doivent convenir d'un checkpoint en consensus et l'insérer de manière uniforme dans le graphe de tâches. Avec StarPU, il n'est pas possible d'insérer un checkpoint entre des tâches déjà soumises car cela nécessiterait de revoir l'ensemble du graphe de tâches soumis ce qui est extrêmement coûteux. Nous pouvons uniquement insérer dynamiquement un checkpoint entre les tâches soumises et celles qui le seront, c'est à dire à la soumission. Il est possible de laisser le runtime convenir d'un checkpoint et l'insérer à un certain point dans la soumission. Les nœuds doivent donc stopper leur soumission jusqu'à un certain point défini dynamiquement, ce qui peut générer des famines de tâches sur certains nœuds. De plus le résultat est dépendant de l'équilibrage de la charge entre les nœuds. Par exemple si un nœud est très en avance dans par rapport aux autres dans sa soumission, on ne pourra pas réaliser de checkpoint avant que l'ensemble des nœuds ne rattrapent leur retard. Choisir d'insérer un checkpoint ralentirait potentiellement le calcul. Le checkpoint pourrait aussi se terminer en réalité bien plus tard que le moment où l'on a choisi de l'insérer, notamment dans le cas où le front de soumission n'évolue

pas de façon homogène par rapport à l'exécution.

Si l'on souhaite s'affranchir du surcoût des communications de consensus et du décalage dû au déséquilibre de charge, on pourrait laisser le runtime choisir de réaliser un checkpoint sur la base d'un critère déterministe, qui pourrait être évalué localement par tout les nœuds et ce de manière uniforme. Définir un tel critère est en pratique difficile et nous n'avons pas voulu nous confronter à cette problématique dans le cadre de cette thèse ; il s'agit néanmoins d'une piste intéressante pour des travaux futurs. De plus pouvoir reprendre l'exécution est bien plus difficile car le point de reprise est bien plus difficile à représenter qu'en le définissant statiquement.

2.1.2 Sauvegarder les checkpoints dans la mémoire des autres nœuds

Étant donné que l'on propose de gérer les sauvegardes à l'aide du support d'exécution, il est possible de mettre en commun les données de sauvegardes et les données nécessaires au calcul. Pour bénéficier de cela il faut que les sauvegardes soient effectuées dans la même mémoire que les données de calcul, à savoir la mémoire volatile. Néanmoins cette mémoire n'est localement pas fiable, étant donné qu'en cas de panne la sauvegarde nécessaire au redémarrage est perdue. On peut cependant utiliser la mémoire volatile d'un autre nœud, qui ne sera pas affectée dans le cas de panne isolée. On dit alors qu'un *nœud d'origine* va stocker ses sauvegardes sur un *nœud backup*. Étant donné que cela est suffisant pour tolérer des pannes isolées correspondant au modèle que nous avons choisi de considérer, nous pouvons nous intéresser à comment réaliser des sauvegardes sur d'autres nœuds de manière efficace, c'est à dire en minimisant l'overhead généré.

Une source d'overhead importante à prendre en considération concerne les communications induites par les sauvegardes. Utiliser un support d'exécution qui gère les communications est très intéressant pour cela, car comme vu en Section 1.3.2, on est en capacité de déterminer quelles données sont déjà présentes sur un nœud. Ainsi si l'on choisit de sauver une donnée sur un nœud qui l'a déjà reçue (pour le fonctionnement de l'algorithme applicatif), le transfert n'aura pas à s'effectuer une seconde fois. Cela est illustré à la Figure 2.6.

En revanche le checkpoint étant sémantiquement inséré dans le code de l'application, on avait pu voir que cela sépare les tâches soumises des tâches qu'il reste à soumettre. Ainsi au moment de soumettre le checkpoint, il est possible de savoir si certaines données que l'on doit sauvegarder sont déjà présentes sur certains nœuds, mais il est impossible de savoir si un nœud va avoir besoin, pour réaliser une tâche qui n'a pas encore été soumise, des données qui doivent être envoyées pour être sauvegardées. Avec ce modèle il n'est donc pas possible de déterminer précisément au moment de la soumission du checkpoint quel est le nœud sur lequel il vaut mieux sauvegarder un checkpoint afin de réduire les communications induites, car les informations sont partielles. Insérer les

checkpoints statiquement apporte donc un autre avantage ici, car cette information peut être connue du développeur applicatif ou déduite par une analyse statique. C'est pourquoi nous proposerons dans l'interface de checkpoint de laisser l'application spécifier quel nœud est le backup du checkpoint, tout simplement car le développeur ou une analyse statique du code a accès à davantage d'informations que le support d'exécution si l'on cherche à réduire les communications induites par les sauvegardes. Cela permet d'atteindre une synergie entre les transferts de données applicatifs et les transferts utilisés pour réaliser les checkpoints, dont on discutera en section 2.3.7.1.

Copy-on-write pour les checkpoints

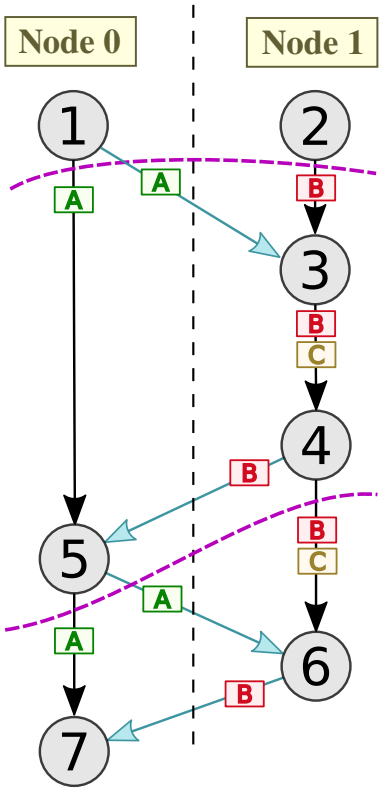
On a vu à la section 2.1.1.2 que lorsqu'une tâche modifie une donnée qui fait partie d'un checkpoint précédent, cette tâche doit attendre que la sauvegarde de cette donnée soit achevée avant de pouvoir écrire dedans. Pour éviter cette synchronisation, il est possible de dupliquer la donnée, et conserver une copie le temps de la sauvegarde pendant qu'une tâche modifie cette donnée.

En partant du principe que le runtime propose un mécanisme inspiré du *copy-on-write* pour les données du calcul, il est possible d'éviter de telles copies. Dans ce cas les données de sauvegarde ne sont pas forcément dupliquées dès que la donnée originale est disponible, mais plutôt dès que la donnée originale va être modifiée. Néanmoins, même si cette dernière solution permet de réduire potentiellement l'encombrement mémoire, il peut être contre-productif de retarder toutes les copies pour sauvegarde au dernier moment car cela retarde potentiellement l'exécution des tâches qui souhaitent écrire dans ces données. Employer cette technique serait donc sensiblement pénalisant si toutes les données que l'on doit sauvegarder sont systématiquement modifiées par des tâches exécutées juste après le checkpoint : on a alors retardé au plus tard une copie qui doit être faite, et ce retard entraîne une latence dans l'exécution. En revanche, si la donnée sauvegardée n'est jamais modifiée ou si la modification intervient bien plus tard, il est alors contre-productif de la copier systématiquement, et l'utilisation d'un mécanisme de copy-on-write devient justifiée. L'efficacité d'un tel mécanisme est donc dépendante de l'application, mais reste intéressant car la latence induite par les copies de données est souvent négligeable.

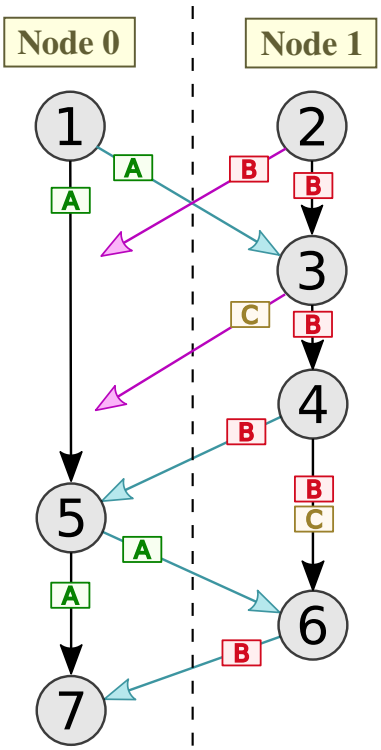
Notons que ce mécanisme de copy-on-write pourrait profiter à StarPU en dehors du contexte de la tolérance aux pannes. Actuellement une tâche de type write-after-read (Section 2.1.1.2) qui a une dépendance en écriture sur une donnée reste bloquée tant que toutes les tâches en lecture sur cette donnée ne sont pas terminées. Il serait alors très certainement intéressant d'effectuer une copie de cette donnée afin de débloquer au plus tôt cette dépendance en écriture. Pour aller plus loin dans des travaux futurs, une analyse statique permettrait de savoir si le copy-on-write est pertinent ou pas, et l'on

```
task_insert(&f1, A, W);
task_insert(&f2, B, W);
checkpoint();
task_insert(&f3, B, RW, A, R, C, W);
task_insert(&f4, B, RW);
task_insert(&f5, A, RW, B, R);
checkpoint();
task_insert(&f6, B, RW, A, R, C, R);
task_insert(&f7, A, RW, B, R);
```

(a) Code source de l'exemple, identique au premier code 2.3a.



(b) Graphe de tâches obtenu avec le code 2.6a.



(c) Transferts effectivement initiés pour le graphe de la Figure 2.6b

FIGURE 2.6 : Illustration de l'impact du choix de nœud de sauvegarde. On a ici décidé de sauvegarder les checkpoints du nœud 0 sur le nœud 1 et inversement. Pour le premier checkpoint, le nœud 0 envoie déjà la donnée A au nœud 1 pour la tâche 3, il n'y a donc pas besoin de la renvoyer pour le premier checkpoint. Le nœud 1 doit par contre envoyer la donnée B au nœud 0 pour le premier checkpoint. Pour le deuxième checkpoint, les données A et B sont déjà envoyées pour les tâches 6 et 7, il n'y a pas besoin de les renvoyer pour le deuxième checkpoint..

pourrait laisser l'application spécifier si le runtime doit employer ce mécanisme pour une sauvegarde.

2.1.3 Redémarrage global ou local ?

Ces deux types de redémarrage sont complémentaires. On aura tendance a utiliser un redémarrage local si un seul nœud tombe en panne, mais le redémarrage global peut être une solution intéressante quand le nombre de nœuds en panne est trop grand. En choisissant de répliquer des sauvegardes uniquement dans les mémoires volatiles, le redémarrage global devient même obligatoire si plusieurs pannes affectent des nœuds simultanément. On peut s'intéresser dans un premier temps au cas du redémarrage global.

2.1.3.1 Redémarrage global pour StarPU

On parle de redémarrage global lorsque l'on redémarre l'ensemble des nœuds depuis un checkpoint global cohérent. Dans notre cas un checkpoint global est l'ensemble des checkpoints locaux réalisés par les nœuds, correspondant à une unique coupure dans le graphe de tâches.

En partant du principe que l'on sauvegarde les checkpoints uniquement dans la mémoire principale des nœuds, il faut effectuer des sauvegardes supplémentaires. Dongara et al. [31] ont montré que chaque nœud doit garder une copie de son propre checkpoint en local afin de garantir la possibilité de survivre à la perte d'un seul nœud. En effet le nœud perdu contenait certainement une partie de la sauvegarde d'autres nœuds qui est définitivement perdue en cas de panne, et alors le checkpoint est incomplet. Si toutes les sauvegardes existent en deux exemplaires, en l'occurrence dans la mémoire d'un nœud backup et la mémoire du nœud d'origine de la sauvegarde, une panne d'un seul nœud est alors récupérable en toutes circonstances du moment qu'elle reste isolée à ce nœud. Pour un panne simultanée de plusieurs nœuds cette méthode peut ne plus suffire, car certains scénarios de panne peuvent rendre inaccessibles toutes les copies des sauvegardes d'un même nœud, en l'occurrence si le nœud backup tombe en panne en plus du nœud d'origine de la sauvegarde. De manière générale on pourrait faire autant de copies d'une sauvegarde que nécessaire, en faisant $n + 1$ copies d'une sauvegarde pour s'assurer de la récupération après une panne simultanée de n nœuds. Néanmoins cette approche est limitée car augmenter le nombre de sauvegardes nécessaires fait augmenter linéairement l'encombrement mémoire et la bande passante consommée par ces sauvegardes.

Si l'on souhaite être sûr de pouvoir reprendre le calcul lors de pannes simultanées de plusieurs nœuds, on préférera dans ce cas utiliser des sauvegardes stockées sur un stockage stable comme le système de fichier du super-calculateur. La littérature regorge

de solutions pour faire cela, et l'on pourra facilement ajouter cette fonctionnalité par la suite. On pourra continuer à faire du checkpoint incrémental, minimisant les effets d'embouteillage sur le stockage stable qui sont observés dans les solutions habituelles. L'autre avantage de stocker sur disque est que l'on peut volontairement interrompre le programme pour le restaurer plus tard si l'on est en présence de contraintes sur l'allocation temporelle de ressources de calcul par exemple, en laissant la possibilité à StarPU de lire des fichiers de sauvegarde au lancement. Terminer l'exécution du programme pour la reprendre peut également représenter un avantage en cas de panne, car cela est parfois plus rapide que de gérer un redémarrage global par les nœuds sans terminer l'application, mais le projet Reinit1.2.3.1 a tendance à inverser la tendance.

Le redémarrage global est parfaitement compatible avec ce qui a été présenté si on sauvegarde en plus les checkpoints sur un support mémoire non volatile. Néanmoins on ne pourra pas optimiser l'approche davantage que ce qui a été présenté jusqu'ici.

2.1.3.2 Redémarrage local pour StarPU

Il est également important de s'intéresser au redémarrage local car il s'agit d'une approche à priori plus efficace quant à la gestion de pannes isolées. En effet cette méthode permet implicitement de ne pas perdre davantage de calcul que ce qui a été effectué par le nœud en panne depuis son dernier checkpoint utilisable. De plus cela est réalisable en utilisant les fonctionnalités apportées par l'extension au standard MPI proposée par l'implémentation ULFM dont nous avons parlé en Section 1.2.4. Cette dernière propose des options pour manipuler les communicateurs MPI en cas de panne d'un nœud, en offrant un système de détection de panne, l'objectif étant de pouvoir continuer l'exécution de l'application sans être forcé de la terminer entièrement pour la redémarrer. Il est ainsi possible de stocker les sauvegardes dans la mémoire principale des nœuds et de bénéficier de plus de réactivité apportée par cela, mais des limites sont tout de même identifiables. Outre la consommation de la mémoire induite par ces sauvegardes, la gestion de pannes simultanées peut être compliquée car l'on peut perdre définitivement une sauvegarde. Stocker une sauvegarde sur plusieurs nœuds peut apparaître comme une solution envisageable, mais il serait dans ce cas préférable d'utiliser alors une solution de redémarrage global comme évoqué précédemment. En revanche plusieurs pannes de nœuds simultanées peuvent tout de même être gérées, à condition que les sauvegardes nécessaires au redémarrage des nœuds perdus n'étaient pas stockées sur ces mêmes nœuds. On peut se rassurer en se disant que les scénarios de pannes les plus probables sont donc couverts par cette approche, d'autant que l'utilisation d'un runtime tel que StarPU à court terme n'est pas de l'ordre de l'exaflop mais plutôt sur des échelles où les taux de pannes sont tout de même plus cléments.

Idéalement on aura à terme une solution de tolérance aux pannes sur plusieurs

niveaux, avec des sauvegardes en mémoire d'autres nœuds pour traiter une panne de nœud isolée, doublé de sauvegardes sur disque pour les pannes multiples en adaptant une fréquence de sauvegarde plus grande, ces pannes étant moins probables, semblable aux approches de checkpoint multi-niveaux vues en 1.2.2.4.

Nous nous intéressons par la suite uniquement au redémarrage local car il s'agit de ce qui nous permet de bénéficier au mieux des apports d'un support d'exécution tel que StarPU. En effet on pourra traiter uniquement les pannes isolées qui sont néanmoins les plus probables, ce qui correspond davantage aux besoins réels à court terme des utilisateurs de StarPU en terme d'échelle. De plus on pourra voir au Chapitre 3 qu'implémenter le redémarrage local dans StarPU n'est pas forcément trivial, et les recherches menées pour proposer quelque chose de fonctionnel étaient plus intéressantes que si l'on avait choisi le redémarrage global.

2.2 Interface et détails d'implémentation

La mise en œuvre des checkpoints au sein de StarPU a amené à effectuer différents choix conceptuels. On en expose ici les plus intéressants.

2.2.1 Interface de définition des checkpoints

La première chose à éclairer consiste à savoir comment reprendre l'exécution. En effet le code d'application sera relancé, et l'on veut qu'il se positionne à l'endroit qui correspond à la reprise. On pourrait imposer à l'utilisateur de ne faire rien d'autre que des appels à StarPU dans son application, et ainsi assurer que le runtime connaisse parfaitement le comportement de l'application. On aurait alors parfaitement connaissance des données d'état de l'application, et l'on pourrait juste ignorer la soumission des n tâches qui précèdent le checkpoint depuis lequel on redémarre. Néanmoins aucune application utilisant StarPU ne respecte cette contrainte actuellement, et instaurer cette règle n'est pas raisonnable.

Il faut par conséquent laisser la charge à l'utilisateur d'indiquer quelles données sont critiques dans la représentation de l'état de son application. Il faut également lui imposer d'instrumenter lui-même le code pour atteindre la bonne instruction lorsque l'on veut redémarrer depuis un checkpoint. Au final l'interface est identique à l'interface que proposent FTI et VeloC, qui a été présentée en Section 1.2.2.3 avec le code 1.4.

On avait dit en Section 2.1.2 que l'on souhaitait pouvoir laisser l'application définir pour chaque nœud, sur quel nœud envoyer le checkpoint. On peut aussi considérer que l'emplacement optimal des sauvegardes peut être différent non seulement selon les données d'un même checkpoint, mais aussi pour une même donnée d'un checkpoint à l'autre. Il est donc important de laisser la possibilité à l'application de définir l'emplacement

pour chaque donnée, mais aussi pour chaque checkpoint.

Nous proposons de définir un *template* de checkpoint, structure contenant des méta-données nécessaires au checkpoint. Lors de la définition d'un template il s'agit d'associer pour chaque donnée nécessitant d'être sauvegardée, le rang MPI du nœud sur lequel l'on souhaite sauver la donnée. Lors de la soumission du checkpoint dans le code de l'application, il est donc possible de choisir le template que l'on souhaite utiliser, qui contient l'information de la distribution des sauvegardes. On peut ainsi définir un unique template et l'utiliser à chaque soumission de checkpoint, et les données seront ainsi sauvées selon la même distribution à chaque checkpoint soumis. Mais il est également possible de définir un template différent pour chaque checkpoint, si la distribution optimale des sauvegardes venait à changer entre chaque soumission. Il suffit dans ce cas de passer le template adéquat lors de la soumission de chaque checkpoint.

Le code 2.1 montre la définition d'un template de checkpoint. La création d'un template se fait en trois parties. La fonction `starpu_mpi_checkpoint_template_create` permet de de créer un template vide. Le développeur doit fournir un `template_id` unique différent à chaque template créé.

Sauver des données gérées par StarPU

On doit ensuite compléter les informations du template, en associant un nœud de sauvegarde à chaque donnée gérée par un handle StarPU. On utilise pour cela la fonction `starpu_mpi_checkpoint_template_add_entry`. La constante `STARPU_R` permet de préciser que l'on ajoute une entrée pour un handle, qui sera accédé en lecture. La distribution de sauvegarde choisie est de sauver les données du nœud i sur le nœud $i+1 \pmod k$. On considèrera que le nœud h est le *backup principal* du nœud i, quand h est indiqué comme backup de la première donnée du nœud i insérée dans le template. Enfin lorsque toutes les données à sauvegarder sont renseignées, on termine la création du template avec `starpu_mpi_checkpoint_template_freeze` qui permet de figer le template pour le rendre utilisable par la suite.

L'interface proposée est intéressante car permet d'avoir la granularité la plus fine qui soit exploitable, à savoir préciser pour chaque donnée où cette dernière doit être sauvegardée. Néanmoins l'identification d'une distribution optimale peut être compliquée pour le développeur étant donné qu'un des buts de StarPU est d'affranchir ce dernier de la charge de suivre les transferts de données. Une analyse statique est donc nécessaire pour vraiment exploiter ce niveau de détail, et un compilateur pourrait se charger de cette tâche, comme nous pourrons le voir dans les travaux futurs en conclusion.

En revanche il est toujours possible de simplifier grandement l'interface en proposant un wrapper qui permette de remplir un template de manière automatique, en lui fournissant simplement une fonction déterministe qui donne le nœud sur lequel une

```
...
int vec[k*n] = {...}; // Vecteur de données de calcul
/* On souhaite distribuer ce vecteur sur k nœuds,
en faisant k sous-vecteurs de taille n. */
starpu_data_handle data_h[k];
starpu_mpi_checkpoint_template_t cp_template;
starpu_mpi_checkpoint_template_create(&cp_template, 0/*template_id*/);
for(int i=0 ; i<k ; i++) { /*On itère sur le nombre de sous données*/
    starpu_vector_data_register(&data_h[i] /*Handle du sous vecteur de taille n*/,
                                STARPU_MAIN_RAM, /*Mémoire contenant les données*/
                                &vec[i*n] /*Pointeur vers données du sous-vecteur*/,
                                n /*Taille du sous vecteur*/,
                                sizeof(int) /* Taille d'un élément */);
    starpu_mpi_data_register(data_h[i],
                    tag, /*Tag qui est utilisé par MPI*/
                    i); /*On affecte data_h[i] au nœud de rang i*/
    starpu_mpi_checkpoint_template_add_entry(&cp_template,
                    STARPU_R,  data_h[i],
                    (i+1)%k /*nœud backup pour la donnée data_h[i]*/ );
}
starpu_mpi_checkpoint_template_freeze(&cp_template);
...
```

Code 2.1 : Exemple de définition de checkpoint template.

donnée est sauvée en fonction du nœud propriétaire de la donnée par exemple. Le runtime pourrait alors appliquer le résultat de cette fonction de distribution à tous les handles déclarés avant l'appel au wrapper, assurant que toutes les données aient un nœud de sauvegarde. De plus à l'avenir, si l'on souhaite ne pas utiliser la sauvegarde sur les autres nœuds mais uniquement celle sur disque, on pourra se passer d'insérer le handle dans le checkpoint template. En effet StarPU peut sauver automatiquement tous les handles, étant donné que ce sont forcément des données critiques de l'état de l'application, tout en conservant les propriété du checkpoint différentiel.

Néanmoins certaines données qui doivent être sauvegardées ne sont pas gérées par StarPU, et l'interface doit permettre de les traiter.

Sauver des données externes à StarPU

Si l'on sait sauver automatiquement les données gérées par StarPU via les handle, les données internes à l'application, et externes à StarPU, doivent en revanche sauvegardées. En partie 1.2.2.1 dans le code 1.2, nous avions vu le genre d'instrumentation de code qui permettent à l'application de ne pas exécuter ce qui doit être ignoré quand on redémarre depuis un checkpoint. L'utilisateur doit donc effectuer cette instrumentation, placer ces données de contrôle dans le `checkpoint_template`, mais aussi toute donnée externe à StarPU qui est représentative de l'état de son programme. Les contraintes

```
...
int i, i_0=0;
starpu_mpi_checkpoint_template_t cp_template;
starpu_mpi_checkpoint_template_create(&cp_template, 0/*template_id*/);
... /* Init des handle StarPU */
 starpu_mpi_checkpoint_template_add_entry(&cp_template,
                    STARPU_VALUE, /*Indique une donnée externe à StarPU */
                    &i_0, /* Pointeur vers la donnée à sauver*/
                    sizeof(i_0), /* Taille de la donnée */
                    0 /* Tag unique */,
                    backup_function /* Fonction de distribution des sauvegardes */ );

starpu_mpi_checkpoint_template_freeze(&cp_template);
...

if starpu_mpi_is_restarted() {        // Est vrai si l'on redémarre après une panne.
    starpu_mpi_restore_checkpoint();  // Modifie les variables protégées par le valeurs
}                                     // du checkpoint.

for (i=i_0 ; i<IMAX ; i++) {  // Itération à partir de i_0
        ...    /* Do some work */
        i_0 = i+1;
        starpu_mpi_insert_checkpoint(cp_template, 0 /*prio*/); // On insère une requête
                                                               // de sauvegarde.
}
```

Code 2.2 : Exemple de sauvegarde des données externes au support d'exécution.

pour l'utilisateur sont exactement les mêmes que ce qu'il aurait dû faire en utilisant d'autres solutions de checkpoints de niveau application.

La première chose à faire est de distinguer que l'on souhaite ajouter une donnée externe à StarPU avec la constante `STARPU_VALUE`. On donne ensuite simplement la donnée à sauvegarder en indiquant son pointeur et sa taille en octets. Un tag est ensuite nécessaire afin d'identifier la donnée au sein du checkpoint, afin de la restituer en cas de panne, et doit par conséquent être unique. Enfin il est important de saisir qu'une donnée externe peut avoir une valeur différente pour chaque nœud à la soumission d'un même checkpoint, même si dans cet exemple simple la variable d'itération évolue de manière identique sur chaque nœud.

La différence est que pour les données de StarPU, les handles nous permettent d'identifier une donnée unique qui est attribuée à un seul nœud. Pour les données externes, avec une programmation SPMD, il existe une version de cette donnée pour chaque nœud. Il faut donc préciser pour chaque nœud quel est le nœud backup pour la sauvegarde de cette donnée. Afin de gagner en accessibilité, on peut fournir à la fonction `starpu_mpi_checkpoint_template_add_entry` une fonction de répartition des sauvegardes avec le prototype `int (*fn)(int)`, qui retourne le nœud de sauvegarde en fonction du nœud qui possède la donnée. Ici cette fonction retourne simplement $i + 1 \pmod{k}$ quand elle reçoit i. Comme pour les autres fonctions de répartition présentées,

il faut que celle-ci soit également déterministe, c'est à dire avoir un comportement identique sur tous les nœuds afin que cela soit cohérent.

On note qu'il serait intéressant de laisser la possibilité au développeur de préciser qu'une donnée externe à StarPU est commune et homogène à tous les nœuds, comme pour le cas de la variable d'`i_0` dans le code 2.2. En effet lors de la soumission de checkpoint à l'itération n, cette variable i vaut la même valeur n pour tous les nœuds, et n'a en principe pas besoin d'être transférée ; on pourrait dans ce cas prendre la valeur locale comme étant la valeur du nœud que l'on doit sauvegarder, plutôt que d'ajouter un transfert qui est dans ce cas évitable.

Insertion du checkpoint

Le checkpoint est inséré dans la soumission avec la fonction `starpu_mpi_insert_checkpoint`. Lorsque StarPU exécutera cette fonction il créera une nouvelle structure de checkpoint, en lui associant un ID incrémental qui démarre à la valeur 1. Uniquement les données externes à StarPU fournies par l'application sont prêtes à ce moment. Elles sont copiées dans des buffers par StarPU à ce moment-là, afin de ne pas rester trop longtemps dans cet appel. Pour les données gérées par les handles, StarPU crée ensuite de nouvelles dépendances sur les données qu'il faut effectivement sauvegarder si la donnée a été modifiée depuis le dernier checkpoint (checkpoint différentiel), et ces données seront automatiquement envoyées. On peut aussi poster dès à présent les réceptions de messages d'acquittement.

L'appel à cette fonction indique que le nœud local doit sauver ses propres données, mais aussi qu'il doit s'attendre à recevoir des données de sauvegarde de la part des autres nœuds. La fonction `starpu_mpi_insert_checkpoint` crée donc également des requêtes de réception afin de correctement traiter les sauvegarde des autres nœuds. La liste des données à recevoir et la liste des nœuds qui vont envoyer ces données sont des informations déterministes connues de chaque nœud localement, et sont définies dans le checkpoint template.

L'argument `priority` dans la fonction `starpu_mpi_insert_checkpoint` permet de fixer une priorité aux communications qui vont être induites par le checkpoint. Si des priorités ont été affectées aux tâches par l'utilisateur, la priorité affectée à un checkpoint permettra à StarPU de déterminer quelle communication effectuer en premier.

On pourra également utiliser plusieurs templates pour répliquer chaque checkpoint sur plusieurs nœuds. Il suffira d'affecter des nœuds de backup différents dans chaque template, et d'insérer les checkpoints en insérant les templates. Par exemple pour répliquer les checkpoints selon deux stratégies différentes, on appellera la fonction `starpu_mpi_insert_checkpoint(cp_template1, cp_template2, prio)`. De cette manière le checkpoint sera effectué selon les deux templates, et sera considéré comme

un seul checkpoint avec un ID incrémenté de 1 par rapport au checkpoint précédent.

En pratique très peu de choses sont faites pendant cette fonction, et l'on sortira très rapidement afin de reprendre la soumission. Les tâches qui sont soumises après le checkpoint n'ont ainsi pas besoin d'attendre que le checkpoint soit terminé pour être exécutées. Cette approche est très efficace car StarPU s'occupera lui-même de sauver chaque donnée quand elle est dans l'état qui est spécifié par le checkpoint. De cette manière on n'a pas besoin que toute l'application soit dans un état cohérent, uniquement les données, et l'une indépendemment de l'autre.

2.2.2 Valider un checkpoint

Pour valider la réception d'une sauvegarde, on ne veut pas reposer sur un acquittement venant du système de communication. On ne peut pas faire cela avec des `MPI_Issend` et `MPI_Isrecv`, pour des raisons propres au fonctionnement interne à StarPU. En effet le nœud backup doit effectuer des actions pour sauver les données, et ce n'est que lorsque ces dernières sont effectuées que l'on peut considérer que la sauvegarde est effectuée.

Le nœud d'origine du checkpoint doit donc s'assurer que chaque donnée à sauvegarder est stockée de manière pérenne. Chaque donnée étant potentiellement sauvée sur des nœuds backup différents, chaque backup doit acquitter la sauvegarde de toutes les données auprès du nœud d'origine. En utilisant le modèle de checkpoint template présenté précédemment, l'ensemble des nœuds concernés par une sauvegarde savent précisément combien de messages de sauvegardes doivent être réalisés pour un checkpoint donné. Il est donc possible d'avoir un unique message d'acquittement par nœud backup, qui indique au nœud d'origine que l'ensemble des données dont il a la responsabilité sont correctement sauvées.

Le nœud d'origine du checkpoint peut collecter les messages d'acquittements et considérer la sauvegarde terminée une fois l'ensemble de ces messages reçus. Le nombre de messages d'acquittement à attendre est déduit avec exactitude grâce au checkpoint template. Le nœud sait désormais que sa sauvegarde est validée et doit désormais partager cette information.

Pour effectuer un redémarrage depuis un checkpoint, celui-ci doit être complet. Les nœuds restants pourront déterminer quel checkpoint utiliser pour redémarrer le nœud dès lors qu'une panne survient. Comme on ne tolère qu'une seule panne, il faudra que l'information du checkpoint valide soit partagée à au moins un autre nœud. Par convention on préviendra le nœud backup principal (Section 2.2.1). Ce choix est totalement arbitraire mais on souhaite que toutes les informations soient envoyées au même endroit pour simplifier certaines synchronisations. On pourrait utiliser une collective si l'on souhaitait partager à tous les nœuds quel checkpoint est valide, mais

ce n'est pas nécessaire dans notre cas.

Pour envoyer ces messages, le thread de progression a été modifié afin de pouvoir laisser StarPU envoyer des données qui ne sont pas des données du calcul. Uniquement StarPU est exposé à cette interface que l'on a appelé messages de service.

2.2.3 Garbage collector

Afin d'éviter que l'empreinte mémoire du checkpoint ne croisse indéfiniment, il est nécessaire de mettre en place un mécanisme de garbage collector. Comme l'on ne suit pas par défaut la progression des checkpoints, chaque nœud n'a pas une idée claire de la progression globale de tous les nœuds. Le checkpoint le plus ancien auquel on est susceptible de redémarrer est un checkpoint qui appartient au dernier checkpoint global complet, et on peut supprimer sans crainte tous les checkpoints qui sont causalement antérieurs au dernier checkpoint global complet. Pour que tous les nœuds connaissent quel est le denier checkpoint global complet, une collective est nécessaire. Cette collective sera exécutée de manière non bloquante et gérée par le thread de progression des communications de StarPU-MPI.

On peut implémenter simplement le garbage collector en faisant en sorte que lorsque leur checkpoint local $N \pmod g$ est validé (avec N l'ID du checkpoint et g une période d'exécution du garbage collector), chaque nœud poste une barrière non bloquante. Quand le test de cette barrière sera vrai, on pourra supprimer tous les checkpoints dont l'ID est inférieure à N. On pourra changer g afin que le garbage collector s'exécute plus ou moins régulièrement pour éviter un encombrement mémoire ou au contraire éviter de trop nombreuses barrières.

On peut sinon exécuter le garbage collector sur demande de StarPU, lorsqu'il a besoin de faire de la place. Un nœud pourrait demander aux autres nœuds, via les messages de service, de lancer une collective permettant d'identifier le dernier checkpoint global complet. Avec une collective `MPI_Iallreduce` et une opération `MPI_MIN`, chacun des nœuds mettra l'ID du dernier checkpoint pour lequel il a reçu une validation. Comme chaque nœud k envoie ses validations de checkpoints sur le nœud $k+1 \pmod N$, on saura exactement quel est le dernier checkpoint global complet lorsque la collective terminera.

2.3 Expériences et résultats

On propose d'évaluer le mécanisme des checkpoints proposés. Nous nous sommes focalisés sur le coût des sauvegardes, car celui-ci permet d'avoir une bonne idée de l'overhead généré dans le cadre d'une exécution sans panne. Le coût du redémarrage est important pour juger de l'efficacité d'un mécanisme de checkpoints, mais n'ayant pas

été implémenté dans StarPU, il ne sera pas étudié ici.

L'objectif de cette thèse est de proposer un mécanisme de tolérance aux pannes qui aille le plus loin possible dans les interactions que l'on peut raisonnablement faire entre un support d'exécution et un protocole de tolérance aux pannes. Les expériences réalisées sont donc avant tout là pour valider notre approche, évaluer les performances étant simplement un bonus.

2.3.1 Problème étudié

Pour évaluer notre approche, nous avons choisi d'étudier un code de décomposition de Cholesky, étant donné qu'il fournit des propriétés de graphe intéressantes, contrairement aux codes itératifs. Ces derniers (Stencils ou gradient conjugués par exemple) ont en effet un comportement très régulier, qui permet de placer les checkpoints beaucoup plus facilement dans le code d'application afin d'avoir une fréquence de sauvegarde homogène. La décomposition de Cholesky produit au contraire un graphe de tâches alambiqué qui restreint les possibilités d'avoir des checkpoints réguliers. Comme on a vu en section 2.1.1.3, la fréquence de checkpoint ne peut pas être imposée, elle est directement induite par l'ordre de soumission des tâches. On essaie donc de soumettre les tâches dans un ordre qui soit intéressant, i.e., qui permette de pouvoir sectionner de manière régulière le graphe de tâches avec les checkpoints. Le placement des checkpoints dans la soumission doit être équilibré afin que les checkpoints progressent le plus régulièrement possible lors de l'exécution, et ainsi avoir une couverture efficace.

2.3.2 Ordres de soumission des tâches pour la décomposition de Cholesky

On souhaite insérer les checkpoints simplement dans le code de l'application. Pour un code comme une factorisation de Cholesky, un moyen simple est d'insérer le checkpoint à la fin de la boucle de soumission principale.

Fréquence de checkpoints

La régularité des checkpoints dépend dans ce cas de l'équilibre de la boucle principale, i.e., du fait qu'elle soumette un nombre de tâches équivalent pour chaque nœud de calcul à chaque itération. On préférera donc choisir un ordre de soumission qui suit l'ordre de priorité d'exécution, mais aussi dont la boucle principale de soumission est équilibrée. Si dans le cas de Cholesky la boucle principale ne peut pas être parfaitement équilibrée, certains ordres de soumissions sont néanmoins plus déséquilibrés que d'autres.

Algorithm 1 STF tile Cholesky, triangle-wise (or right-looking)

```
for (k = 0; k < NT; k++) do
  task_insert(&POTRF, RW, A[k][k]);
  for (m = k+1; m < NT; m++) do
    task_insert(&TRSM, R, A[k][k], RW, A[m][k]);
  for (n = k+1; n < NT; n++) do
    task_insert(&SYRK, R, A[n][k], RW, A[n][n]);
    for (m = n+1; m < NT; m++) do
      task_insert(&GEMM, R, A[m][k], R, A[n][k], RW, A[m][n]);
  checkpoint();
task_wait_for_all();
```

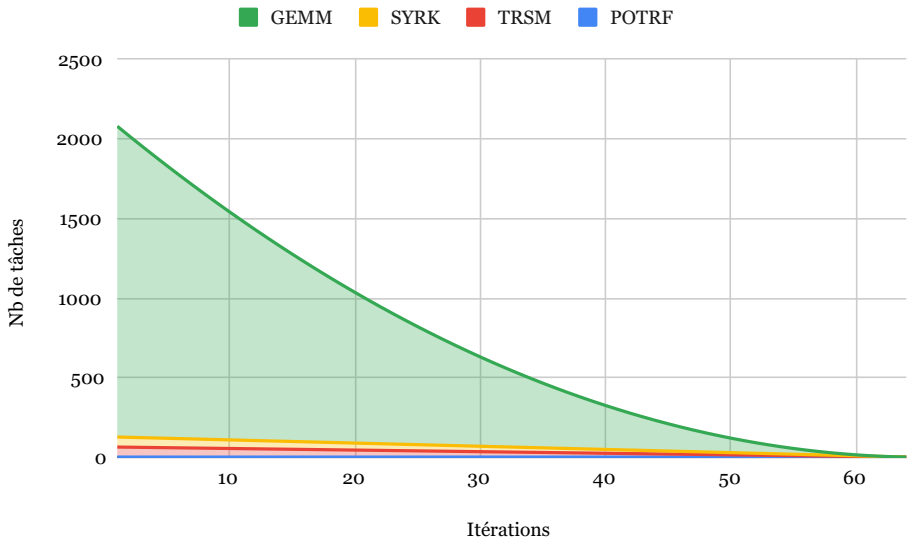


FIGURE 2.7 : Nombre de tâches soumises à chaque itération avec l'algorithme 1(triangle-wise ou right-looking), pour une matrice carrée de $N_t = 64$ tuiles de large

La soumission standard de Cholesky soumet les tâches par sous-matrices triangulaires, comme décrit dans l'algorithme 1 ; cette soumission est connue des numériciens sous l'appellation *right-looking*. Ce qui émerge de cet ordre de soumission est que chaque itération de la boucle principale soumet de moins en moins de tâches. Pour une matrice de N_T tuiles de large, l'itération i de la boucle principale (on pose $i = k + 1$, k allant de 0 à $N_T - 1$) soumet 1 tâche `POTRF`, $N_T - i$ tâches `TRSM`, $N_T - i$ tâches `SYRK` et $N_t(N_t - 1) - i(i + 2N_t - 1)$ tâches `GEMM`. L'évolution du nombre de tâches soumises à chaque itération est représenté sur la Figure 2.7. On voit que les tâches `GEMM` sont les plus nombreuses dans un graphe de tâches de Cholesky, et le fait qu'elles suivent une tendance N_tš $- i$š, et donc diminuent de manière quadratique à chaque itération, crée un fort déséquilibre dans la boucle principale. Si nous effectuons des checkpoints toutes les x itérations avec ce code de soumission, les checkpoints risquent de ne pas

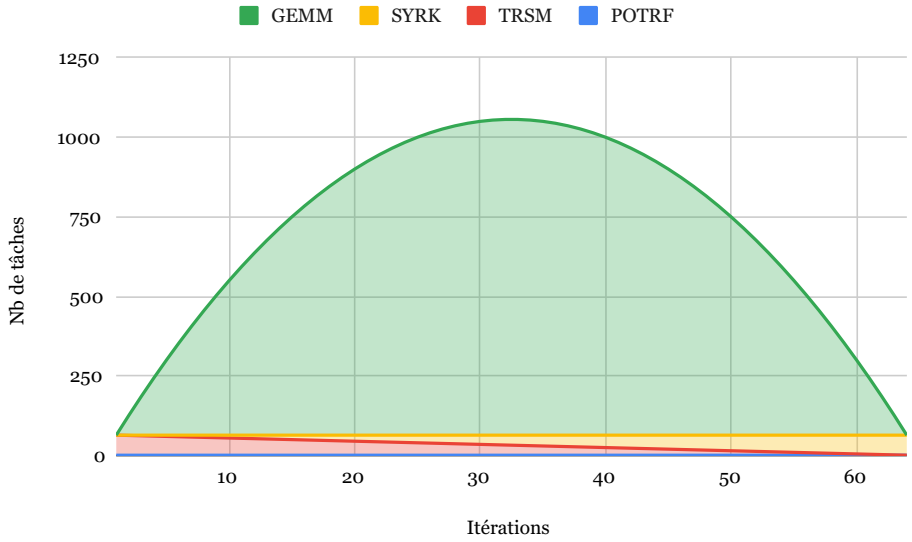


FIGURE 2.8 : Nombre de tâches soumises à chaque itération avec l'algorithme 2 (column-wise ou left-looking), pour une matrice carrée de $N_t = 64$ tuiles de large

être bien répartis temporellement : ils seront de plus en plus fréquents à mesure que l'exécution progresse, étant donné que la quantité de travail entre chaque checkpoint diminue avec le temps. On peut interpréter la politique de cette soumission avec la proposition suivante : tout en ne soumettant qu'un seul `POTRF` à chaque itération, on soumet toutes les tâches qui peuvent alors être soumises.

Algorithm 2 STF tile Cholesky, column-wise (or left-looking)

```
for (n = 0; n < NT; n++) do
  for (k = 0; k < n; k++) do
    task_insert(&SYRK, R, A[n][k], RW, A[n][n]);
  task_insert(&POTRF, RW, A[n][n]);
  for (m = n + 1; m < NT; m++) do
    for (k = 0; k < n; k++) do
      task_insert(&GEMM, R, A[m][k], R, A[n][k], RW, A[m][n]);
    task_insert(&TRSM, R, A[n][n], RW, A[m][n]);
  checkpoint();
task_wait_for_all();
```

Si l'on soumet le graphe avec l'algorithme de soumission 2, la boucle principale est plus équilibrée. Le principe de cette boucle est de soumettre à l'itération k toutes les tâches qui écrivent une tuile de la colonne k. On soumet donc les tâches par colonnes, contrairement à l'algorithme précédent qui soumettait par sous-matrice triangulaire. Cette ordre de soumission a pour effet de mieux répartir les tâches `GEMM` et `SYRK` dans les itérations, alors que les tâches `POTRF` et `TRSM` sont soumises aux mêmes itérations

que précédemment. L'itération i (avec $i = n + 1$) soumet 1 tâche POTRF, $NT - i$ tâches TRSM, $i - 1$ tâches SYRK et $(N_t - i)(i - 1)$ tâches GEMM. On commence par remarquer sur la Figure 2.8 que le nombre de tâches TRSM et SYRK est constant et vaut $N_t - 1$ quelle que soit l'itération, alors qu'il était décroissant avec l'algorithme 1. Ces tâches ayant une complexité similaire, la charge reste équilibrée à toute les itérations concernant ces tâches. Le nombre de TRSM n'est pas constant mais n'est plus strictement décroissant en fonction de l'itération. On voit que la boucle principale de soumission de l'algorithme 2 est plus relativement mieux équilibrée que celle de l'algorithme 1. En insérant des checkpoints toute les x itérations, on s'attend donc à ce que les checkpoints soient mieux équilibrés au cours l'exécution. On devrait avoir des checkpoints mieux répartis dans le temps, avec tout de même un phénomène de dilatation de la fréquence des checkpoints au milieu de l'exécution, les tâches étant soumis en plus grand nombre. On notera tout de même qu'il est possible de moduler la fréquence d'apparition des checkpoints dans le code d'application, si leur fréquence à l'exécution s'avère trop disparate. En particulier on pourra facilement forcer l'application à soumettre plus de checkpoints en milieu de soumission, afin de répartir davantage la charge de travail entre chaque checkpoint. On peut interpréter la politique de cette soumission avec la proposition suivant : tout en ne soumettant qu'un seul POTRF par itération, on ne soumet que les tâches qui permettent de compléter entièrement une tuile à la fin de l'itération.

Taille des checkpoints

L'ordre de soumission choisi n'impacte pas uniquement l'équilibre du nombre de tâches entre les checkpoints ; il impacte également le coût des checkpoints. La quantité de données qu'il faut sauver lors d'un checkpoint est toujours la même : on doit sauver toutes les données définies dans le template utilisé. Mais comme l'on réalise des checkpoints de manière différentielle (en ne sauvant que les données qui ont été modifiées depuis le dernier checkpoint), le coût d'un checkpoint dépend de son contenu, selon s'il a plus ou moins évolué depuis le dernier checkpoint. On avait vu en Section 2.1.1.1 que le contenu d'un checkpoint inséré dans un code SPMD est déterministe. Le contenu dépend donc uniquement de son emplacement dans le code d'application ; et les emplacements possibles dans lesquels on peut insérer un checkpoint dépendent directement de l'ordre de soumission choisi.

De ce fait le coût total des checkpoints va être différent selon l'ordre de soumission choisi, pour un même nombre de checkpoint.

L'algorithme 1 soumet à chaque itération des tâches qui modifient toute les tuiles qui n'ont pas atteint leur valeur finale. On peut voir sur la Figure 2.9 que la première itération modifie la matrice entièrement ; la première colonne atteint sa valeur finale et ne sera plus modifiée par les tâches soumises aux itérations suivantes. Le premier

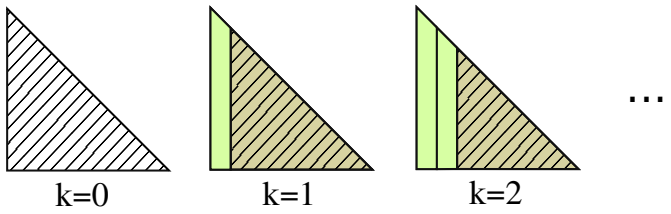


FIGURE 2.9 : Accès des tâches soumises avec l'algorithme 1 en fonction de l'itération. Les zones hachurées sont les zones accédées par les tâches à l'itération k. En blanc : les zones à l'état initial au début de l'itération. En vert : zones à l'état final en début d'itération. En marron : zones à l'état intermédiaire en début d'itération.

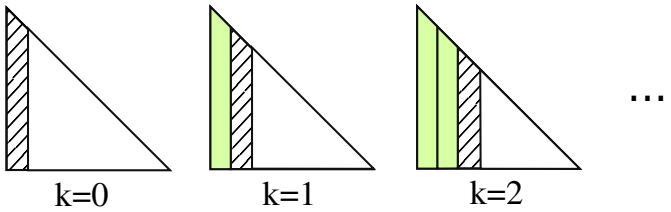


FIGURE 2.10 : Accès des tâches soumises avec l'algorithme 2 en fonction de l'itération. Les zones hachurées sont les zones accédées par les tâches à l'itération k. En blanc : les zones à l'état initial au début de l'itération. En vert : zones à l'état final en début d'itération. Il n'y a pas de zone en état intermédiaire en début d'itération.

checkpoint devra donc sauver l'intégralité de la matrice, car elle est entièrement modifiée dès la première itération. En revanche à la fin l'itération i, les i premières colonnes ne seront plus modifiées dans le reste du calcul et les futurs checkpoints ne dupliqueront pas inutilement cette valeur. Avec cet ordre de soumission, on sauve des valeurs de calcul intermédiaires qui sont modifiées à chaque nouvelle itération. Ainsi même si l'on utilise un garbage collector efficace pour supprimer les anciens checkpoints, l'empreinte mémoire totale des checkpoints sera donc toujours supérieure à la taille de la matrice, et est directement dépendante du nombre de checkpoints effectués.

En revanche, l'algorithme 2 amène un comportement différent qui est très intéressant pour nous. En effet en soumettant les tâches par colonne, l'itération i soumet toutes les tâches qui écrivent dans la colonne i et uniquement celles-ci. Ainsi, comme on peut le voir sur la Figure 2.10, on est sûr qu'à la fin de l'itération i la colonne i a atteint son état final tandis que les colonnes à partir de $i+1$ ne sont pas encore modifiées. Par conséquent, quelle que soit l'itération à laquelle apparaît le checkpoint, la matrice à sauver ne contient que des données qui sont soit à l'état initial, soit à l'état final. Avec notre checkpoint différentiel, StarPU ne sauve pas les données qui sont à l'état initial et le checkpoint ne contient donc que des données à l'état final. Comme on ne sauve aucune donnée intermédiaire, l'empreinte mémoire totale des checkpoints est bornée quel que soit le nombre de checkpoints. Ici, cette borne est définie par la taille de la matrice, contrairement à ce que l'on avait avec l'algorithme 1, pour lequel cette borne

est dépendante du nombre de checkpoint.

Ces deux ordres de soumission, lorsque l'on attribue des priorités aux tâches afin de respecter le chemin critique, amènent à des performances similaires. Mais dans notre cas il est préférable d'utiliser l'algorithme de soumission 2 pour exploiter les propriétés que l'on vient de voir. Dans la Section 2.3.7.1, nous reviendrons sur les propriétés permettant d'identifier si pour d'autres application, il est possible d'avoir un ordre de soumission qui autorise un tel comportement.

2.3.3 Plateforme

Les expériences ont été réalisées avec 25 nœuds Miriel de la plateforme d'expérimentation PlaFRIM. Chaque nœud est composé de 2 Intel® Xeon® E5-2680 v3 Haswell 12 cores @ 2.5 GHz et de 128 Go de mémoire (5.3 Go/core, @2 933 MHz). L'interconnexion est réalisée via un réseau Infiniband 40 Gb/s. Les noyaux de calculs utilisent MKL2019 avec AVX2 et l'implémentation MPI est OpenMPI 4.0.3. La performance moyenne des tâches `GEMM` avec des tuiles de taille 320×320 est de 60.2 GFlop/s (simple précision) par cœur, ce qui donne une performance maximum théorique de 36.12 TFlop/s pour les 600 cœurs disponibles. Nous avons fait les mesures en utilisant la politique d'ordonnancement de StarPU "lws" (local work stealing). Cette politique répartit les tâches prêtes de manière équilibrée sur les ressources de calcul disponibles. Il n'y pas d'accélérateurs sur ces machines, les ressources sont donc uniquement les cœurs de processeurs. Quand une tâche est terminée, les tâches qui en dépendent sont à priori exécutées sur le même cœur ; si le cœur n'a plus de tâche prête à exécuter, il vole des tâches aux autres cœurs, en récupérant en priorité les tâches associées aux cœurs qui sont le plus proche dans l'architecture de la machine. Cette politique est efficace pour éviter les concurrences d'ordonnancement, en pipelinant au mieux les tâches sur les cœurs, et ce même avec un grand nombre de cœurs.

2.3.4 Overhead en performance

Nous avons étudié uniquement le coût des checkpoints, ce qui permet d'avoir une idée des performances à attendre d'une exécution sans panne. La Figure 2.11 montre les performance de l'application en fonction de la dimension de la matrice, selon différents nombres de checkpoints soumis dans l'application. L'application est la factorisation de Cholesky soumise par colonne avec l'algorithme 2. Les checkpoints de chaque nœud sont sauvés dans la mémoire principale d'un autre nœud de calcul, la politique utilisée étant volontairement simpliste : le rang k envoie ses checkpoints au rang $k + 1 \pmod{25}$. La distribution des tuiles se fait avec une répartition 2D Block Cyclic 5×5.

La première chose à remarquer est qu'en augmentant la taille de la matrice, l'overhead relatif a tendance à diminuer alors que la taille des checkpoints augmente. Ce

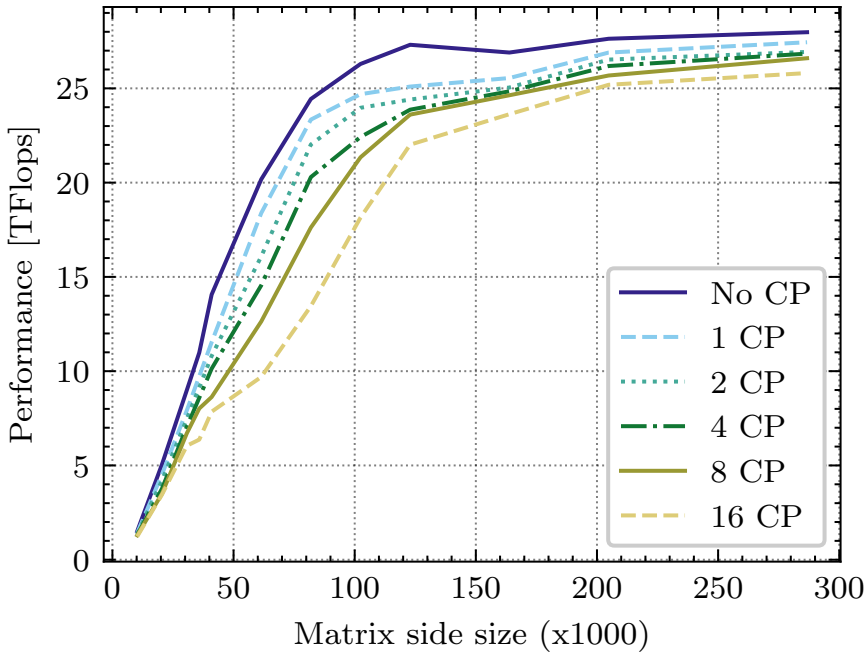


FIGURE 2.11 : Performances de Cholesky avec l'algorithme 2 (soumission par colonne) sur 25 nœuds MPI en fonction de la largeur de la matrice en nombres flottants simple précision, pour différents nombres de checkpoints.

phénomène est principalement dû au fait que le temps d'exécution augmente avec la taille du problème, et que donc pour un nombre de checkpoints fixé, la période entre les checkpoints augmente. Dans ces mesures, les checkpoints ont en fait une fréquence temporelle qui est bien plus élevée que ce que l'on utiliserait en pratique. Pour la plus grande matrice étudiée (286 720 × 286 720 soit 896 tuiles de largeur), le calcul original dure en moyenne 4 min 40. Avec 16 checkpoints uniformément répartis dans la boucle de soumission, l'exécution dure en moyenne 5 min 4s (+24 s, soit +8.6 %) avec une période de checkpoint moyenne de 19 s. L'overhead est en revanche plus agressif pour de plus petites matrices ; avec une matrice carrée de taille 384 tuiles de largeur (122 800 × 122 800 soit environ 4× moins de données), le calcul original dure 22.5 s et effectuer 16 checkpoints prend 5.7 s de plus que l'exécution sans tolérance aux pannes. L'overhead est alors de 25 %, mais s'explique par la période de checkpoint moyenne qui est de 1.76 s.

On avait pu voir qu'avec la soumission de l'algorithme 2, l'empreinte totale des checkpoints est bornée et ne dépasse pas la taille originale de la matrice. Le coût des checkpoints est donc borné quel que soit le nombre de checkpoints, cette borne évoluant en $\frac{n^2}{2}$, alors que le temps de calcul d'une décomposition évolue en n^3, n étant la largeur de la matrice. Le fait que l'overhead diminue quand la taille du problème augmente s'explique aussi par cela. Et donc en dépit de n'avoir étudié le comportement

sur uniquement 25 nœuds, le passage à l'échelle semble envisageable avec ces propriétés. Il devrait en être de même pour les applications qui peuvent bénéficier des mêmes propriétés.

Optimisations possibles

La façon dont étaient implémentés les checkpoints dans StarPU était sous-optimale lors des mesures. En effet le copy-on-write n'est pas encore implémenté dans StarPU. À chaque fois qu'une nouvelle donnée est reçue pour un checkpoint, elle est copiée systématiquement (si elle ne fait pas partie d'un checkpoint précédent). Une donnée de calcul est ici une tuile de taille 320×320, donc la durée d'une copie en elle même n'est pas problématique (de l'ordre de la centaine de microsecondes). Néanmoins la copie est effectuée avec une tâche, et va donc monopoliser une ressource de calcul. Étant donné que l'on crée autant de tâches qu'il y a de tuiles à copier, cela peut avoir un effet considérable. Il faudra donc implémenter le copy-on-write, qui nous permettra de n'effectuer la copie que si nécessaire.

On peut aussi noter que l'on a associé aux checkpoints une priorité, qui dépendait de la tâche la plus prioritaire soumise lors de la même itération. En pratique, les communications d'un checkpoint n étaient plus prioritaires que les communications pour des tâches proches du chemin critique qui ont été soumises après le checkpoint n. Les communications du checkpoint ont donc eu pour effet de ralentir la progression de l'exécution, alors que la bande passante consommée était loin des limites imposées par le réseau. Pour diminuer cet effet, on pourrait affecter une priorité plus faible aux checkpoints. Dans ce cas on aura une meilleure performance, mais cela veut également dire que la complétion des checkpoints sera potentiellement retardée. Il faudra donc fournir une priorité adéquate aux checkpoints, afin de trouver un bon compromis entre favoriser les performances et favoriser la qualité de couverture.

2.3.5 Overhead en communications

On peut voir dans le Tableau 2.1 la quantité de données transférées par nœud spécifiquement pour les checkpoints, pour une matrice carrée de de 380 tuiles de large ($121\,600 \times 121\,600$). Dans nos conditions, la décomposition de Cholesky induit en moyenne 10.28 Go de données envoyées par nœuds pour les seuls besoins applicatifs. Le pourcentage entre parenthèses indique la bande passante consommée en plus de cela pour les checkpoints. Ce tableau compare plusieurs approches afin d'identifier les bénéfices réalisés.

La colonne "Checkpoint aveugle" est la quantité de donnée qui est transférée lorsque l'on ne peut pas suivre les modifications de données. On est dans ce cas forcé de sauver

#CP	Checkpoint aveugle (% additionel)	+ différentiel (% additionel)	+ cache de comms. (% additionel)	+ distribution maline
1	1.18 Go (+11.5%)	0.892 Go (+8.67%)	0.239 Go (+2.33%)	∼ Ko
2	2.37 Go (+23.1%)	1.055 Go (+10.3%)	0.423 Go (+4.11%)	∼ Ko
4	4.74 Go (+46.1%)	1.143 Go (+11.1%)	0.622 Go (+6.05%)	∼ Ko
8	9.49 Go (+92.2%)	1.175 Go (+11.4%)	0.776 Go (+7.58%)	∼ Ko
16	18.98 Go (+184%)	1.185 Go (+11.5%)	0.888 Go (+8.64%)	∼ Ko

TABLE 2.1 : Quantités moyennes d'envois de données par nœud, induits par les checkpoints pour une matrice carrée de 380 tuiles de large, pour différents nombre de checkpoints. Code de Cholesky soumis avec l'algorithme 2 (soumission par colonne) sur 25 nœuds MPI.

toutes les données protégées, même si elles sont toujours à l'état initial ou inclues dans un checkpoint précédent. Ce n'est pas notre cas avec StarPU, ces données sont théoriques et utilisées à titre de comparaison. Pour cette colonne, chaque checkpoint sauvegarde donc tout ce que l'application a inclus dans les données à sauvegarder, dans notre cas cela représente toute la matrice triangulaire, soit 29.57 Go. Chaque nœud envoie en sauvegarde sa contribution dans la distribution de la matrice, $1/25^{\text{ème}}$ de la matrice dans notre cas, soit 1.18 Go. On remarque donc que la quantité de données transférées est juste le produit entre cette taille et le nombre de checkpoints.

La colonne "+ différentiel" indique la quantité de données transférées lorsque l'on ne sauve que les données qui diffèrent des checkpoints précédents. Chaque nœud suit les modifications de données effectuées par lui-même, mais aussi celles des autres nœuds. On sait dès la soumission du checkpoint si une donnée a été modifiée depuis l'état initial ou depuis le dernier checkpoint. Quand c'est le cas, la donnée n'a pas besoin d'être sauvegardée étant donné qu'elle l'est déjà. Comme vu en Section 2.3.2, la soumission par colonne permet de créer des checkpoints qui ne contiennent que des données soit à l'état initial, soit à l'état final. C'est ici que l'on voit que l'empreinte mémoire totale des checkpoints est alors bornée par la taille de la matrice quel que soit le nombre de checkpoints. On remarquera que pour un checkpoint placé en milieu de soumission, la taille du checkpoint représente la partie gauche de la matrice triangulaire, qui vaut donc $3/4$ de la matrice triangle.

La troisième colonne montre les données effectivement transférées lors des mesures de la Figure 2.11. On bénéficie ici du checkpoint incrémental, mais aussi de la réplication induite par le calcul, en se servant du cache de communications de StarPU-MPI. On avait vu en Section 1.3.2 que le runtime n'effectue qu'une seule communication lorsque plusieurs communications identiques sont redondantes. Quand une donnée est déjà envoyée, on ne la renvoie pas car l'émetteur et le récepteur reconnaissent la redondance sur des critères locaux et déterministes. Le nombre de communications par rapport à la colonne précédente est plus faible, car certaines données du checkpoint sont en fait déjà

transférées par le calcul.

La dernière colonne indique les quantités de données transférées effectivement si l'on répartit les sauvegardes de manière optimale. Dans nos expériences, on a appliqué une répartition des sauvegardes au plus proche, en sauvegardant le rang k sur le rang $k+1 \mod 25$, alors que la distribution des données suit une répartition 2D Block Cyclic 5×5. Certains nœuds qui font la sauvegarde des autres ne communiquent donc jamais au cours du calcul, n'étant jamais sur une même ligne ou une même colonne. Si l'on avait réparti les sauvegardes plus intelligemment, en précisant que le nœud k a pour nœud backup $k - (k \mod 5) + ((k+1) \mod 5)$ par exemple (c'est-à-dire le nœud suivant sur la même ligne de la répartition 2D bloc-cyclique), on aurait eu une synergie complète avec le calcul. Avec cette répartition des sauvegardes, aucune donnée du checkpoint n'est jamais envoyée spécifiquement pour la sauvegarde, étant donné que le nœud backup a déjà besoin de la donnée pour les calculs applicatifs ; ainsi les nœuds ne sauvegardent pas leurs données sur des nœuds qui n'avaient pas déjà besoin de la donnée pour l'application. Ainsi la bande passante consommée est réduite à quelques kilo-octets correspondant aux données externes à StarPU, dans ce cas les données de contrôle telles que la variable d'itération par exemple, et les messages de service utilisés par notre protocole (pour l'acquittement et la validation). Néanmoins ce résultat est un cas particulier obtenu avec un ordre de soumission, une distribution des sauvegarde optimisés et en exploitant des propriétés spécifiques à une décomposition de Cholesky ; on ne pourra pas attendre de tels résultats avec n'importe quelle application. Par exemple, avec la soumission de l'algorithme 1 le résultat n'aurait pas été le même étant donné que l'on sauvegarde des données intermédiaires qui ne sont jamais propagées en temps normal, et on aurait ainsi eu de multiples envois de données spécifiques aux checkpoints.

On peut aussi remarquer que dans le cas de Cholesky, les données de calcul sont toujours envoyées à au moins deux autres nœuds (sauf pour les deux dernières tuiles du calcul). On pourrait alors indiquer dans le template plusieurs nœuds backup si l'on souhaite répliquer la sauvegarde sur plus d'un nœud. Si l'on choisit bien ses nœuds backups, on peut là aussi faire des sauvegardes multiples qui ne génèreront pas de communications supplémentaires pour autant.

2.3.6 Progression des checkpoints

Comme vu en Section 2.1.1.1, la progression des checkpoints est complètement asynchrone. Une donnée d'un checkpoint qui doit être sauvée l'est immédiatement après la complétion de la tâche qui la produit. Dans notre cas, on initie le transfert vers le nœud de sauvegarde à ce moment là. On n'attend donc pas que toutes les données du checkpoint soient prêtes pour initier la sauvegarde, ce qui dilue la mise en œuvre du checkpoint au cours du temps. Aussi on discutait en Section 2.3.2 de l'impact du

placement des checkpoint dans leur occurrence lors de l'exécution.

On peut voir sur la Figure 2.12 le comportement lors des mesures réalisées. Le début et la fin de chacun des 12 checkpoints (numérotés de 1 à 12) sont représentés sur ce graphe. Le début est l'instant où la première donnée est envoyée en sauvegarde, et la fin est le moment où l'on enregistre l'information que le checkpoint est complet. On voit que pour la plupart des checkpoints, les données commencent à être sauvées avant même que le premier checkpoint ne soit terminé. Cela est dû au fait que des tâches sur le chemin critique sont exécutées mais apparaissent plus tard dans la soumission, et donc appartiennent à des checkpoints éloignés. Comme la tâche est exécutée en priorité, la donnée est prête tôt dans l'exécution et la sauvegarde est initiée en conséquence. Cet effet est néanmoins limité aux tâches proches du chemin critique, les autres tâches suivant plutôt l'ordre de soumission par colonne.

On peut noter que l'on s'attendait à avoir des checkpoints rapprochés en début d'exécution avec cet ordre de soumission alors que la période a finalement une tendance décroissante, sans que ça ne soit très problématique pour autant. Ceci est dû au fait que les priorités des tâches proches du chemin critique retardent l'exécution de certaines tâches ; par exemple les tâches `SYRK` des tuiles des dernières lignes de la première colonne sont retardées lors du premier checkpoint. L'ordre de soumission par triangle de l'algorithme 1 n'est pas montré ici, mais nous avons observé une progression des checkpoints finalement assez similaire, avec tout de même une tendance décroissante plus prononcée pour l'intervalle de temps entre checkpoints.

Si l'on avait soumis les tâches avec un algorithme les soumettant par ordre de priorité, on aurait eu un comportement plus régulier : les checkpoints auraient été moins dilués au cours de l'exécution, étant donné que les tâches soumises à la même itération auraient alors été exécutées dans un laps de temps plus resserré. Ce type de soumission est intéressant car provoque une tendance à naturellement minimiser la quantité de travail perdu en cas de panne. Comme l'on fixe le contenu des checkpoints statiquement, et qu'ils sont réalisés de manière asynchrone, quand un checkpoint est terminé on n'a en réalité sauvé qu'une partie du travail réalisé au moment du checkpoint : celui qui a été soumis avant le checkpoint. C'est en fait le prix à payer quand on fait des sauvegardes asynchrones sans stopper l'exécution. La quantité de travail réalisé mais non sauvegardé lorsqu'un checkpoint termine dépend donc aussi de la soumission, comme on avait pu en discuter en Section 2.1.1.3.

2.3.7 Interprétation et discussions

2.3.7.1 Sauvegarder dans les autres nœuds

Sauver les checkpoints sur les autres nœuds n'est pas toujours efficace. En effet, plusieurs conditions doivent être réunies afin d'assurer que l'overhead reste minime.

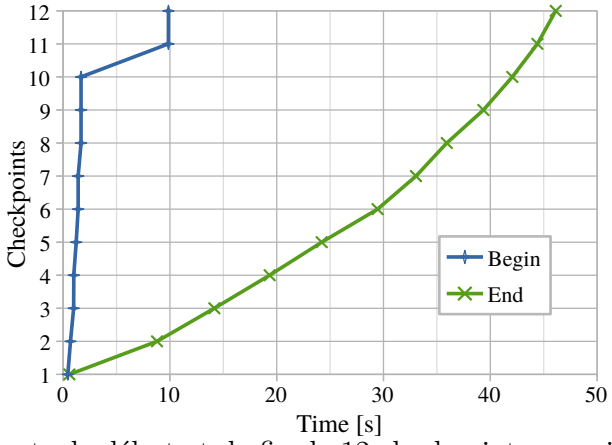


FIGURE 2.12 : Instants de début et de fin de 12 checkpoints soumis uniformément au cours de la soumission de Cholesky avec l'algorithme 2(soumission par colonne)

On a vu que pour une décomposition de Cholesky, il est possible avec cette approche d'avoir des sauvegardes presque gratuites, les données étant déjà envoyées sur d'autres nœuds par le calcul. Cela est possible car dans un graphe de tâches de Cholesky, pour certaines distributions de données initiales, il existe des sous-graphes qui ont pour données initiales des données qui sont naturellement répliquées sur d'autres nœuds, et ce sont ces données que nous utilisons pour constituer un checkpoint. Si dans le graphe de tâches d'une application, on cherche et l'on trouve une distribution de données initiale qui autorise l'existence de tels sous-graphes, alors il est envisageable effectuer des sauvegardes presque gratuites ; il suffira de distribuer les données des checkpoints sur les nœuds qui les possèdent déjà pour le calcul.

Un autre point de vue est de chercher à découper un graphe de tâches en sectionnant des arêtes, où pour chaque arête coupée correspondant à une donnée, il existe une autre arête correspondant à la même donnée qui est également coupée ; on peut alors choisir une distribution des données initiale qui induit que ces paires d'arêtes mènent à des tâches qui sont exécutées sur deux nœuds différents. Dans le cas d'une application de type *stencil*, cela est possible en distribuant les tuiles avec une répartition 2D Block Cyclic, plutôt qu'une répartition par bloc. Mais au final, ce n'est pas intéressant car on rompt la localité qui est induite par la répartition en bloc, les sauvegardes gratuites étant obtenues en augmentant énormément les communications induites par le calcul, ce qui en réalité va détruire les performances. Il faut donc que la répartition des données soit également efficace pour les performances.

Avec l'algorithme 2, on a également une autre propriété qui permet d'être plus efficace en performance : certains des sous-graphes induits par les checkpoints ont pour données initiales des données à l'état final qui ne sont plus modifiées dans le reste du calcul. Cela traduit en réalité que l'état interne de l'application peut être décrit avec peu de données car quelle que soit la taille de tuile, on peut faire le calcul tuile par tuile

car il existe toujours une tuile qui peut être calculée de son état initial jusqu'à son état final sans calculer les autres. Cela permet de faire des points de reprise qui ne reposent pas sur des états intermédiaires, uniquement sur des états de données soit finaux soit initiaux. Étant donné que des sous-graphes de Cholesky regroupent les deux propriétés, on peut avoir une répartition des données qui est à la fois efficace pour le calcul et pour les sauvegardes. Dans un stencil, on ne peut pas calculer de tuile indépendamment d'autres tuiles à l'état intermédiaire, ce qui fait que l'empreinte mémoire des checkpoints ne peut pas être réduite. Il n'existe donc pas de distribution qui permette d'atteindre cette propriété. Comme un stencil peut se calculer en faisant en sorte que les nœuds ne partagent qu'une petite partie de leur état interne, et qu'il est d'autant plus performant quand les transferts sont petits face aux données représentant l'état interne d'un nœud, on ne peut pas rendre notre approche efficace pour ce type d'application.

On peut également noter que plus le nombre de sous-graphes est grand, plus il sera possible de placer de checkpoints. Il faut alors un ordre de soumission qui permette d'insérer les checkpoints correspondant aux sous-graphes, ce qui nécessite de réécrire le code de soumission. Mais faut-il encore pouvoir insérer les checkpoints correspondant aux sous-graphes dans le code soumission, ce qui peut nécessiter de changer l'ordre de soumission des tâches ; dans ce cas il faudra s'assurer que le chemin critique reste prioritaire. De nouveau, pour réduire au maximum la perte de travail en cas de panne, on pourra faire en sorte que l'ordre de soumission suive au mieux l'ordre de priorité des tâches.

Lorsque l'on ne peut pas trouver les propriétés précédentes dans une application, on peut étudier le coût de nos checkpoints. Le coût d'un checkpoint n est relatif à la relation $C_n - T_n$, où C_n est la quantité de données modifiées et qui diffèrent du dernier checkpoint $n-1$, et T_n est la quantité de données de l'état interne qui est transférée par les besoins du calcul. La valeur de C_n est bornée par l'empreinte mémoire totale de l'état interne. On pourrait réduire cette quantité en effectuant un checkpoint à chaque fois qu'une donnée est modifiée, mais une telle fréquence n'est pas raisonnable. On préférera attendre que C_n atteigne sa borne, car on aura dans ce cas un coût linéaire en fonction du nombre de checkpoints. La valeur de T_n peut être modulée en changeant la répartition des données initiale, mais cela peut engendrer une réduction des performances car l'on va réduire les effets bénéfiques de localité. On pourra alors choisir d'insérer les checkpoints uniquement aux endroits où la relation $C_n - T_n$ est minimale, à une fréquence qui permet un overhead acceptable. Comme les checkpoints sont faits en parallèle de l'exécution, seule une partie de ces coûts sera visible en pratique, qui dépendra d'à quel point le calcul est ralenti par la bande passante consommée par les transferts des checkpoints. Pour des applications comme des stencils, il sera très probablement possible d'avoir un meilleur overhead par checkpoint en choisissant de les sauvegarder sur un autre support.

Trouver ces propriétés n'est pas trivial, mais peut être effectué avec des outils d'analyse de code. Il est possible de chercher les sous-graphes pré-cités avec des outils adaptés, des outils de partitionnement de graphes pourraient être exploitables en soumettant des contraintes adaptées. Si les sous-graphes existent, plus ils seront nombreux, plus on pourra insérer de checkpoints et donc améliorer la qualité de couverture et avoir l'opportunité de moduler la fréquence de checkpoint. Pour changer l'ordre de soumission on peut utiliser, par exemple dans le cas d'une factorisation de Cholesky, un compilateur source-to-source utilisant une analyse polyédrale, et il en sera de même pour d'autres applications ayant un code de soumission plus ou moins affine.

Donc pour résumer si une application peut être calculée de manière progressive tuile par tuile, c'est à dire qu'il existe toujours une tuile qui puisse être calculée entièrement sans devoir calculer partiellement une autre tuile, il existe des points de reprise qui induisent des checkpoints ayant une taille totale bornée, car il existe une progression du calcul où l'état du calcul évolue peu. Si cette borne croît moins fortement que la complexité du calcul quand le domaine augmente (n^2 contre n^3 dans le cas de Cholesky), on a la certitude que le passage à l'échelle est possible. Si de plus, certaines tuiles calculées sont nécessaires pour entamer d'autres calcul on peut faire en sorte que les données des checkpoints soient déjà répliquées par les besoins du calcul. Il faut néanmoins que les points de reprise soient suffisamment nombreux pour que l'on ait l'opportunité de sauvegarder régulièrement l'avancement. Pour les applications qui n'ont pas cette propriété, il faudra trouver un compromis :

— S'ils existent, on pourra chercher des points de sauvegarde qui sont majoritairement composés de données à l'état final ou à l'état initial ;
— On peut parfois améliorer la synergie entre les données de sauvegarde et les checkpoints en changeant la distribution des données initiales, en faisant attention à ne pas trop perdre l'effet de localité ;
— On peut analyser le coût des différents checkpoints possibles et choisir de ne soumettre que ceux qui sont les moins chers.

2.3.7.2 Checkpoints asynchrones dans StarPU

Ces expériences ont pu valider le fait que les sauvegardes sur d'autres nœuds ont un intérêt, mais il reste des domaines d'application où l'on ne peut pas exceller ainsi avec l'approche proposée seule. Ce qui émerge aussi de ces expériences est que l'on a validé notre capacité à réaliser des checkpoints cohérents de manière asynchrone. Et le coût demandé à l'utilisateur pour le faire n'est au final pas plus élevé qu'avec une autre approche de checkpoint de niveau application, étant donné que les interfaces sont très proches. La sauvegarde sur les autres nœuds demande tout de même une information en plus que les autres approches ne demandent pas (sur quel nœud sauvegarder). Dans

CP Lvl	Approche classique	Approche proposée
3	Copie sur stockage stable	Copie sur stockage stable
2	Copie sur disque local	Copie sur disque local
1	Copie en mémoire locale (Copy-on-write)	Copie en mémoire locale d'un autre nœud

TABLE 2.2 : Niveaux de sauvegarde des checkpoints exploitables dans StarPU

une approche de checkpoints multi-level, on peut voir notre approche comme le niveau primaire d'une hiérarchie parallèle à la hiérarchie habituelle, comme montré dans la Table 2.2. L'interface proposée est suffisante pour permettre d'implémenter à l'avenir les autres niveaux de sauvegardes. Et si l'on souhaite ne pas effectuer de sauvegarde sur les autres nœuds, alors l'interface sera identique aux autres approches de checkpoint de niveau application.

Chapitre 3

Redémarrage local et message logging

Dans cette partie nous proposons de compléter la contribution précédente afin de réaliser un système de tolérance aux pannes qui puisse effectuer un redémarrage local. Nous avions conclu que les checkpoints tels que décrits précédemment sont suffisants pour réaliser une solution de tolérance aux pannes avec une reprise globale, sous réserve que les sauvegardes soient déplacées de la mémoire principale des nœuds vers un stockage non volatile avant le redémarrage. Néanmoins on peut considérer dommageable le fait de redémarrer des nœuds parfaitement fonctionnels et de perdre tout le travail effectué depuis le dernier checkpoint s'il est possible d'effectuer une reprise plus raffinée.

Nous nous intéressons donc ici à la reprise locale, une solution permettant de redémarrer uniquement le nœud en panne tandis que les autres nœuds continuent leur exécution. En cas de panne isolée d'un nœud, la mémoire principale des autres nœuds n'est donc pas perdue et il n'est donc pas nécessaire de transférer les checkpoints vers un support non volatile. Cela est arrangeant car l'on avait vu que l'efficacité apportée par notre solution de checkpoint vient du fait qu'ils sont stockés dans la mémoire principale des nœuds, et l'on préférerait ne pas avoir à effacer cette mémoire. Cependant les checkpoints ne permettent pas de proposer un redémarrage local à eux seuls, c'est pourquoi nous allons le compléter avec un système de message logging.

3.1 Problématique du redémarrage local

Reprendre une exécution localement à un nœud est à priori plus intéressant que de redémarrer l'ensemble des nœuds. S'occuper de redémarrer uniquement le nœud en panne permet de minimiser la perte du travail accompli : la reprise ne pouvant se faire que sur le dernier checkpoint, tout le travail accompli depuis cette sauvegarde est perdu lors du redémarrage. Si l'on n'a pas d'autres choix que de redémarrer un nœud

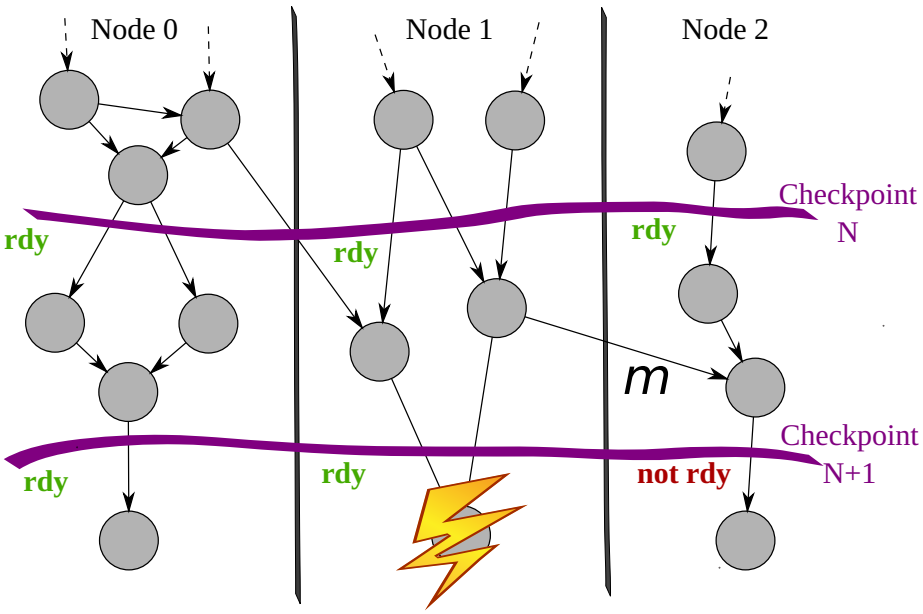


FIGURE 3.1 : Une panne survient sur le nœud 1 alors que le nœud 2 n'a pas terminé son dernier checkpoint.

ayant subi une panne, il est dommage de perdre la progression de nœuds qui peuvent continuer leur exécution. De plus notre système de sauvegarde étant asynchrone, la disponibilité d'un checkpoint global est conditionnée à la progression du nœud le plus lent. Sur la Figure 3.1, le checkpoint global complet le plus récent est celui constitué des checkpoints locaux N, car le nœud 2 est en retard comparé aux autres. Et en cas de déséquilibre de charge, il est possible de n'avoir de disponible uniquement un checkpoint global relativement ancien, ce qui nécessite de jeter davantage de travail pour les nœuds en avance.

Effectuer une reprise locale permet de redémarrer le nœud avec un checkpoint local terminé, même si le checkpoint global associé n'est pas encore terminé. Il faut néanmoins vérifier que redémarrer le nœud avec ce checkpoint ne crée pas d'incohérence. Sur la Figure 3.1, on peut voir que le message m est peut-être un message *en-transit*. En effet, étant donné que les sauvegardes et les envois sont effectués de manière asynchrone, il est possible que le nœud 1 ait terminé son checkpoint N+1, sans que le message m n'ait été effectivement reçu par le nœud 2, et il faut dans ce cas redémarrer le nœud 1 depuis le checkpoint N. En revanche si c'est le nœud 0 qui tombe en panne, il n'y a pas de message en-transit et l'on pourra redémarrer le nœud 0 au checkpoint N+1. Si l'on avait fait un redémarrage global, on aurait dû obligatoirement redémarrer sur le checkpoint global N, car c'est le dernier complet.

Avec un redémarrage local, on peut redémarrer avec un checkpoint plus avancé pour peu qu'il soit cohérent avec les autre nœuds. On voit que le redémarrage local a un intérêt bien réel, et mérite d'être approfondi. Néanmoins, on verra que cela nécessite

de complexifier davantage le protocole de tolérance aux pannes en comparaison d'un redémarrage global.

3.1.1 Que se passe-t-il si un seul nœud est redémarré ?

Lorsqu'une panne se produit, le nœud incriminé est remplacé par un nouveau. Les nœuds ayant contribué à la sauvegarde du dernier checkpoint complet du nœud en panne envoient au nouveau nœud les données du checkpoint afin que celui-ci s'initialise correctement. Ce dernier reprend donc la soumission du graphe de tâches à l'endroit du checkpoint et reprend une exécution normale. Les nouvelles tâches soumises sont des tâches qui ont été soumises par le nœud en panne par le passé, potentiellement terminées, mais dont le résultat a été perdu. Ces tâches, qui sont donc re-soumises du point de vue de la progression globale du calcul, peuvent nécessiter de recevoir des données de la part d'un autre nœud, tel qu'illustré à la Figure 3.2. Quand cela arrive on se retrouve face à un problème : si l'envoi des données a déjà été effectué par le nœud émetteur avant la panne, de son point de vue la donnée est envoyée. On a donc un phénomène de message *en-transit*, qui sera *perdu*. Il se peut de plus que le nœud 1 ait entre-temps perdu l'accès aux données envoyées dans ces messages si elles ont été modifiées dans son flux d'exécution, ou tout simplement si elles ont été supprimées pour libérer de l'espace mémoire dans le cas où elles ne sont plus utiles dans la suite du calcul.

On a donc des données potentiellement manquantes. Avec StarPU, il est très difficile pour les nœuds survivants d'évaluer quels messages doivent être renvoyés afin de faire progresser le nœud redémarré. Comme le runtime ne garde pas de trace du graphe des tâches terminées, il faudrait relire le code de soumission à partir du checkpoint auquel le nœud en panne a été réinitialisé, afin de savoir quoi renvoyer. Et si ces données ne sont plus disponibles, on doit les recalculer en effectuant une tâche déjà terminée, nécessitant des données qui elles aussi peuvent ne plus être disponibles. On se rend alors compte que l'on est en présence d'un effet domino, et l'on peut aller jusqu'à devoir récupérer des données dans son propre checkpoint pour ré-effectuer des tâches. Outre la complexité de revenir en arrière pour StarPU, on peut voir que chercher à recalculer les données manque d'efficacité et il serait plus pertinent d'effectuer un redémarrage global.

On remarque donc que les checkpoints sont insuffisants pour faire un redémarrage local. Il faudrait un système capable d'identifier quels messages doivent être renvoyés à un nœud redémarré, afin pour ne pas avoir à re-parcourir le graphe des tâches (ce que l'on ne peut pas faire avec StarPU sans redémarrer l'application). De plus le contenu de ces messages peut devenir inaccessible au cours de l'exécution, il faudra donc veiller à ce qu'ils soient conservés.

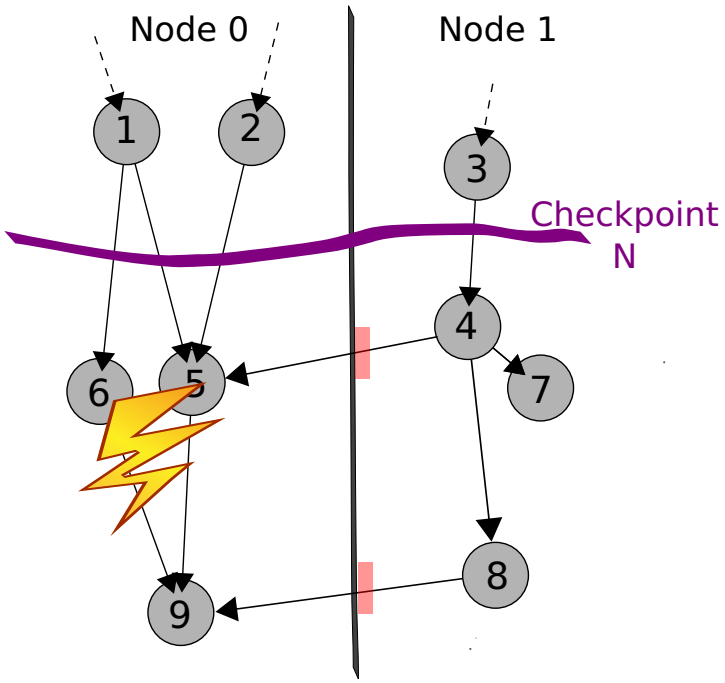


FIGURE 3.2 : Exemple d'envoi de données potentiellement perdu : le nœud 1 a potentiellement déjà envoyé au nœud 0 le résultat de la tâche 4, voire déjà supprimé ce résultat. Pour pouvoir redémarrer le nœud 0, il faudrait alors reprendre une donnée depuis le checkpoint N et ré-exécuter la tâche 4. Pour éviter cela, on peut sauvegarder la donnée au moment de l'envoi, pour pouvoir la renvoyer lors du redémarrage.

3.1.2 Quelles données doivent être conservées ?

On vient de voir que lors d'un redémarrage local, on peut être amené à devoir renvoyer un message que l'on a déjà envoyé. Étant donné que n'importe quel nœud est susceptible de tomber en panne et de voir sa progression revenir en arrière, tous les messages émis par un nœud seront potentiellement nécessaires dans au moins un scénario de panne. On doit donc être en capacité de ré-émettre n'importe quel message émis. Comme le nœud redémarré soumet le même graphe de tâches qu'avant la panne, il aura besoin des même données que lors de l'exécution initiale.

Il faut aussi garantir que l'ordre dans lequel sont ré-émis les messages corresponde bien à l'ordre d'une exécution normale. Là aussi, comme le nœud redémarré soumet le même graphe de tâches qu'avant la panne, le comportement de la réception sera le même que lors d'une exécution normale. On peut donc garantir que les messages seront bien reçus en cas de redémarrage, du moment que les nœuds qui doivent renvoyer des messages les renvoient dans l'ordre auquel ils ont été émis la première fois.

3.2 La solution du Message Logging

Le Message Logging [4] est une solution qui permet de répondre à la problématique précitée. Son but est de conserver une copie de tous les messages et leur contenu, afin de les ré-émettre si besoin. Si des solutions toute prêtes existent, dans notre cas il n'est pas possible de les utiliser. En effet ces solutions reposent sur le fait que les séquences d'envois sont les mêmes d'une exécution à l'autre, et donc déterministes du point de vue de MPI. Avec StarPU cette hypothèse est vraie uniquement si l'on démarre tous les nœuds depuis l'état initial ou depuis un checkpoint global cohérent. Mais dans le cas d'un redémarrage local ce n'est plus vrai : la présence du cache de communication rend indéterministe la séquence d'envoi des messages du point de vue de MPI.

3.2.1 Principe du Message Logging

Le principe du Message Logging tel qu'évoqué dans l'état de l'art est d'effectuer une copie systématique du contenu des messages, ce que l'on peut éviter dans notre cas. Si l'on gère le protocole à l'intérieur de StarPU-MPI, on effectuera une sauvegarde systématique des informations du message, comme le destinataire ou le tag par exemple, mais le contenu du message ne sera pas systématiquement copié. On préfèrera enregistrer dans le log de message un handle vers la donnée concernée, et ainsi déléguer à StarPU le choix de faire une copie de cette donnée si cela est nécessaire. Comme il s'agit davantage de s'assurer que les données soient conservées par les nœuds pour être potentiellement ré-émises, on peut faire juste en sorte que le runtime ne les évince pas lorsqu'il les considère ordinairement comme inutiles. Et dans le cas où la progression de l'exécution

exige de modifier une donnée, un mécanisme de copy-on-write est suffisant pour conserver une copie de la valeur originelle.

3.2.2 Ordre du log de message

Il faut prêter une attention particulière à s'assurer que l'ordre des communications rejouée respecte celui d'un flux d'exécution normal. Avec MPI, les messages envoyés par un nœud à un autre sont délivrés dans l'ordre d'émission, à condition que ces messages aient le même tag. Mais l'ordre n'est plus garanti pour des tags différents. Avec StarPU-MPI, on a un comportement différent mais similaire. On avait vu en 1.3.3.1, que le thread de progression des communications de StarPU-MPI assure une propriété FIFO pour l'ensemble des requêtes de communication qui lui sont soumises (priorités mises à part).

Ainsi on peut se servir de l'ordre dans lequel sont soumises les requêtes au thread de progression pour ordonner le log de messages. En ordonnant le log de messages dans cet ordre exact, on a la garantie que la causalité des messages est respectée. Il suffira donc d'insérer les requêtes dans le log de message dans le même ordre qu'elles sont traitées par le thread de progression des communications. De cette manière le log de message sera correctement ordonné pour un éventuel renvoi.

3.2.3 Sectionnement du log de message

Afin d'identifier clairement les messages à ré-émettre en cas de panne, on propose de sectionner le log de message. En effet quand une panne se produit, il ne faut pas ré-émettre l'ensemble des messages, mais seulement ceux qui sont causalement postérieurs au checkpoint qu'il va utiliser pour redémarrer. Dans notre cas, vu que les checkpoints sont insérés statiquement et de manière uniforme dans le graphe de tâches, les checkpoints divisent déjà le graphe de tâches en sections. Il faut donc sectionner le log de message de la même manière que le graphe de tâches est sectionné avec les checkpoints, tel qu'illustré à la Figure 3.3.

Quand un nouveau checkpoint est soumis dans le graphe de tâches, on sait que toutes les futures tâches soumises seront causalement postérieures au checkpoint. Il en est donc de même pour les communications induites par ces tâches. La section du log dans laquelle placer un message dépend donc de la section du graphe dans laquelle se trouvait la tâche qui a causé la communication. Comme l'information de la section dans laquelle se trouve une tâche n'est disponible qu'à la soumission, il faudra veiller à ce qu'elle soit correctement propagée entre la soumission d'une tâche et le moment où l'on enregistre le message dans le log lors de l'exécution.

Par conséquent on remarque qu'il y a presque autant de checkpoints que de sections dans le log de message. En réalité il y a une section de plus que le nombre de checkpoints,

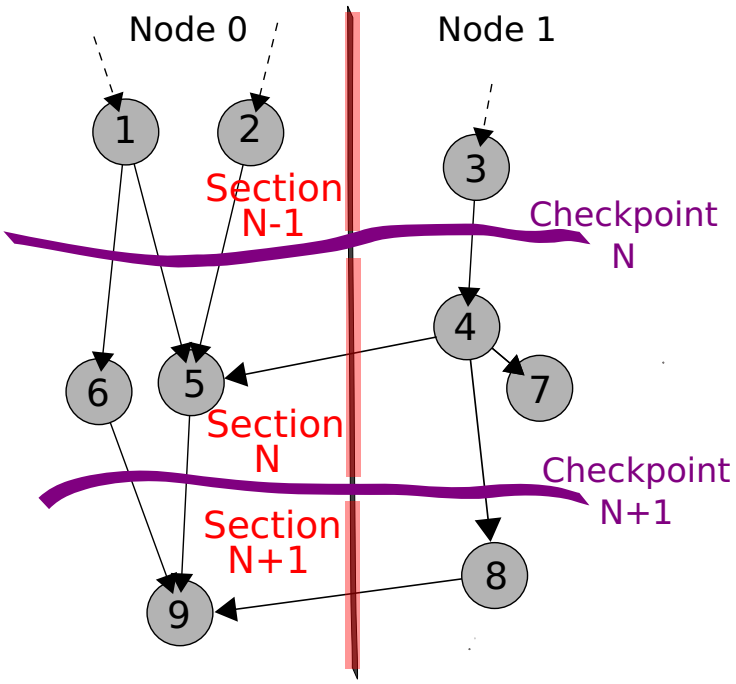


FIGURE 3.3 : Le log de message est sectionné selon les checkpoints.

la section supplémentaire étant celle qui précède le premier checkpoint. Comme l'ID des checkpoints est un compteur qui commence à 1, on peut identifier la première section par la valeur 0. Les futures sections prendront toutes la valeur de l'ID du dernier checkpoint soumis.

Ainsi grâce au modèle STF/SPMD, on peut identifier sans ambiguïté quelle section du log de messages devra être utilisée en cas de panne, en connaissant simplement l'ID du checkpoint utilisé pour le redémarrage. Il n'y a aucun besoin de savoir si le nœud auquel on envoie des données a terminé son checkpoint pour déduire la section dans laquelle placer le message.

On voit que l'on utilise l'ID du dernier checkpoint soumis comme une horloge de Lamport. En effet un message soumis à la section n indique qu'il est causalement postérieur au checkpoint n, mais aussi causalement antérieur au checkpoint $n+1$.

3.2.4 Filtrer les messages sortant du nœud redémarré

Il est important de s'intéresser aux messages émis par un nœud qui redémarre dans un contexte de redémarrage local. En effet un autre phénomène problématique se produit. Ce nœud va se réinitialiser au dernier checkpoint, et exécuter les tâches exécutées avant la panne mais dont le résultat a été perdu. Entre la prise du checkpoint et la panne, ce nœud aura potentiellement envoyé des données à d'autres nœuds, qui auront déjà reçu ces messages comme illustré sur la Figure 3.4. Après redémarrage au dernier checkpoint, le nœud enverra alors de nouveau ces données. Sémantiquement,

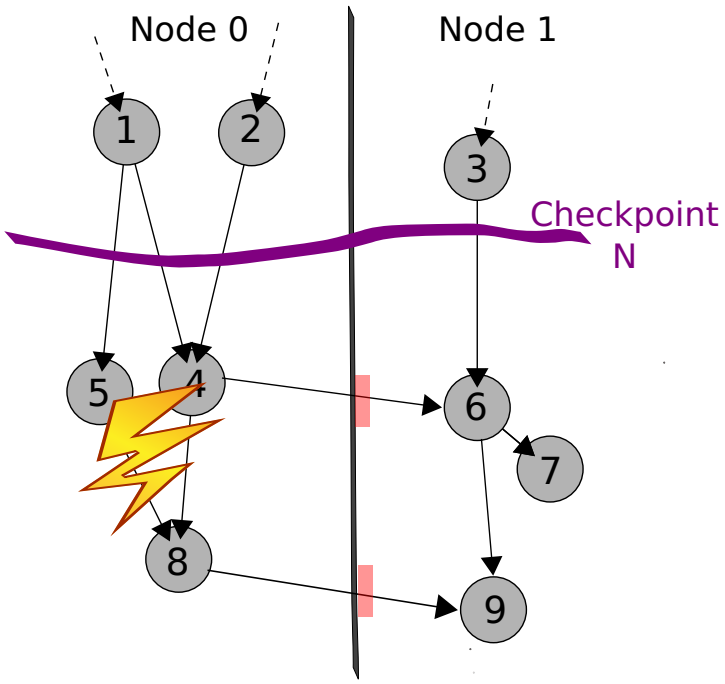


FIGURE 3.4 : Exemple de messages émis par un nœud 0 tombant en panne. Le nœud 1 doit empêcher le nœud 0 de ré-émettre des données. Pour un tag donné, le nœud 1 peut compter le nombre de messages déjà reçus dans la section N, et envoyer les compteurs de chaque tag au nœud 0.

on est face à des messages *orphelins* car le nœud 1 considère que le message a été reçu alors que le nœud 0 considère qu'il n'a pas été envoyé. Les récepteurs ne doivent surtout pas recevoir ces données une seconde fois, car le système entrerait alors dans un état incohérent, i.e., qui ne peut pas arriver lors d'un exécution normale. On peut donc soit choisir d'envoyer les messages mais de les ignorer du côté récepteur, soit forcer le nœud redémarré à ne pas les ré-émettre. Quelle que soit la solution choisie, ces messages doivent pouvoir être correctement identifiés.

Si l'on choisit d'ignorer la réception des messages, on peut définir la liste de ces messages en gardant un log de messages reçus. Ce log de message n'a pas besoin de conserver le contenu des messages, mais uniquement les traces de communication en se basant sur les tags de StarPU-MPI. Sur la Figure 3.4, le nœud 1 a reçu deux messages avec un tag t en provenance du nœud 0 avant qu'il ne tombe en panne. Il suffira donc d'établir des compteurs de messages reçus, pour chaque section de log, pour chaque rang émetteur, et pour chaque tag. Mais cela présuppose que l'ordre des messages sera strictement identique à l'exécution originale pour un tag donné. Il se trouve que StarPU-MPI vérifie cette propriété (à condition que la cohérence du cache soit maintenue, comme nous verrons en Section 3.2.5). L'ordre d'envoi des messages peut différer d'un exécution à l'ordre, mais uniquement pour les messages qui sont causalement indépendants. Pour plusieurs envois successifs d'une même donnée (et donc avec un même tag), l'ordre est

statiquement déterminé par l'application grâce au modèle STF. On a donc la garantie que le nœud redémarré renverra les messages d'un même tag dans la même séquence qu'à l'exécution originale. Pour savoir les messages qui ne doivent pas être renvoyés, les nœuds survivants peuvent donc se baser sur le nombre de messages reçus pour un tag donné dans la section du log de messages concernée. Le nœud redémarré pourra envoyer ses messages naïvement, et le récepteur ignorera les messages orphelins en ignorant le nombre adéquat de messages pour chaque tag.

Si l'on souhaite que le nœud redémarré ne renvoie pas ces messages, les nœuds survivants peuvent lui envoyer le nombre de messages qui doivent être ignorés pour chaque tag. Dans ce cas il faudra que le nœud redémarré ait reçu tous les compteurs avant de commencer à envoyer des messages. On pourra juste forcer le thread de progression des communications à mettre en attente les envois de message tant que tous les compteurs ne sont pas reçus. Ainsi l'envoi des messages orphelins sera évité du côté émetteur. Cette solution est préférable car cela permet de limiter la bande passante consommée, le message d'information ne représentera en effet très probablement qu'une fraction de la taille des messages qui sont évités. Nous avons donc choisi cette approche, qui est inspirée des travaux de Losada et al. [50].

3.2.5 Cohérence du cache de StarPU

Il faut également s'assurer que StarPU se comporte en lui-même de la même manière que dans l'exécution originale vis-à-vis des autres nœuds. En effet un nœud redémarré a perdu des informations de l'état de StarPU qui s'avèrent critiques quand on effectue un redémarrage local, mais qui ne le sont pas pour un redémarrage global. Il s'agit du cache des communications. En effet c'est le seul élément de StarPU qui peut changer le comportement induit par la soumission, et qui rend le nombre d'envois et de réceptions non consistant entre une exécution depuis l'état initial et une section redémarrée.

On peut voir sur la Figure 3.5 que lors d'une exécution normale, le nœud 0 a envoyé la donnée A au nœud 1 avant le checkpoint pour la tâche 2. Lorsque la tâche 3 est soumise, la donnée A n'est pas renvoyée grâce au cache des communications. Le nœud 0 ne soumet donc pas l'émission et le nœud 1 ne soumet pas la réception. Si le nœud 0 tombe en panne il perd son cache de communication, et lorsqu'il soumettra de nouveau la tâche 3, il déduira qu'il faut envoyer la donnée. On a donc encore un message orphelin, considéré reçu par le nœud 1 mais non envoyé par le nœud 0. On peut faire en sorte que le nœud 0 ne renvoie pas cette donnée au nœud 1 en utilisant les compteurs évoqués juste avant (Section 3.2.4). Pour cela il faut qu'à chaque fois qu'une requête de réception est soumise mais évitée par le cache de communication, elle soit tout de même insérée dans le log des messages reçus. Il faut que ce soit une requête différente de celles utilisées lorsque la donnée est effectivement envoyée, car on

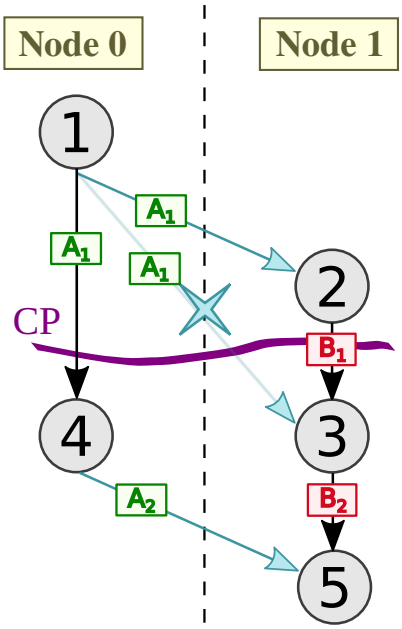


FIGURE 3.5 : Effet du cache de communication sur le redémarrage local.

veut pouvoir la distinguer des requêtes de réception normale. En effet, lors du comptage des réceptions que l'on a déjà reçues, il ne faudra la prendre en compte que sous une condition : si la requête qui a effectivement reçue la donnée est causalement antérieure au checkpoint. Ceci peut être évalué dans StarPU en effectuant quelques modifications, et nous y reviendrons en Section 3.3.4.1.

Par contre si le nœud 1 tombe en panne, un autre problème se produit. Il doit recevoir cette donnée car il en a besoin pour le calcul ; néanmoins elle ne sera ni dans le checkpoint, ni dans le log de messages du nœud 0, car ce dernier considère que le nœud 1 l'a déjà reçue. On a donc un phénomène de message en transit, qui est considéré comme envoyé par le nœud 0 mais non reçu par le nœud 1. Le moyen le plus simple de ré-émettre ce message est de l'insérer dans le log de messages envoyés. De la même manière que précédemment, à chaque fois qu'une requête d'émission est soumise mais évitée par le cache, on insère dans le log de messages envoyés une requête spéciale différente des envois vraiment effectués. Elle est encore différente car lors d'une panne il ne faudra l'envoyer effectivement que sous une condition : si la requête qui a effectivement envoyé la donnée est antérieure au checkpoint. Ceci peut être aussi évalué dans StarPU en effectuant quelques modifications, et nous y reviendrons en Section 3.3.4.1.

3.2.6 Rejouer les communications des checkpoints

Après une panne, le nœud redémarré va soumettre de nouveaux checkpoints. Cela veut dire qu'il cherchera à envoyer des données aux nœuds désignés comme backups de sa sauvegarde, mais aussi s'attend à recevoir des données de sauvegarde de la part des checkpoints dont il est le backup. De plus des messages de service comme les messages d'acquittement et de validation doivent aussi être rejoués. Une partie de ces messages ont déjà été émis et reçus par les nœuds survivants. Il faut s'assurer d'avoir un comportement cohérent.

La première chose à noter est que les communications de données gérées avec des handles sont mises en commun avec les transferts applicatifs et sont donc déjà inclues dans le log de message. Pour les données externes à StarPU, l'utilisateur a associé chaque donnée à un tag unique, afin de pouvoir utiliser le même moyen de transfert que les données gérées par StarPU. Celles-ci sont donc elles aussi correctement traitées par les logs de messages déjà définis. En revanche les messages de service tels que les messages d'acquittement et de validation ne sont pas inclus dans les logs de messages car ils utilisent un protocole de communication différent de celui utilisé pour transférer les données.

Chaque nœud survivant va itérer sur les checkpoints soumis à partir du checkpoint N, N étant le checkpoint choisi pour redémarrer, et vérifier chacun des checkpoints. Si le nœud est le backup du nœud redémarré pour une des données, il va falloir peut-être ré-emettre un acquittement. Si l'on a reçu l'ensemble des données de ce checkpoint, il faudra ré-émettre un acquittement. Les données déjà reçues ne seront pas renvoyées grâce aux compteurs du log de message. Si l'on n'a pas reçu toutes les données il n'y a rien à faire, on enverra l'acquittement en temps voulu.

Les autres messages n'ont pas besoin d'être traités. En effet l'ensemble des nœuds ont en permanence des réceptions sur les messages de service d'acquittement et de validation. Comme ces messages contiennent l'ensemble des informations qui leur permettent d'être traités correctement, on peut avoir des duplications de message sans corrompre le comportement interne de StarPU.

3.2.7 À propos de l'impact sur l'empreinte mémoire

On comprend rapidement que conserver davantage de données en mémoire principale pour constituer un log de message, peut poser des problèmes. Mais notre approche nous permet d'être plus efficaces que les protocoles de messages logging habituels. En effet ces derniers sauvegardent systématiquement les contenus des messages, car ils ne peuvent pas savoir quand la donnée envoyée est modifiée ou supprimée par l'application. Un runtime comme StarPU nous permet d'avoir la main sur ces informations et on peut en tirer avantage. Il suffit juste d'indiquer au runtime que la donnée doit être conservée, et

il la copiera uniquement si nécessaire avec un mécanisme de copy-on-write, tel qu'illustré sur la Figure 3.6.

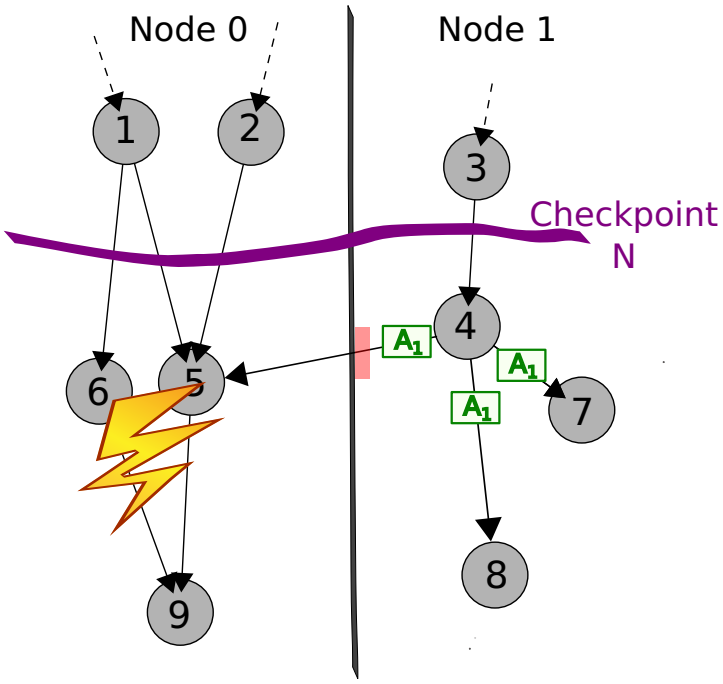
Ainsi notre protocole de message logging revient juste à remplir une table dans laquelle on renseigne quelle donnée est envoyée à qui. La latence induite sur les communications est donc négligeable, en particulier face aux protocoles qui effectuent la sauvegarde des messages avant le transfert.

De plus en évitant la copie systématique on limite une consommation mémoire excessive quand c'est possible. En effet avec notre approche on s'assure juste que les données du log de message restent disponibles au cours de l'exécution, en faisant des copies uniquement quand nécessaire. Une donnée contenue dans un message n'a pas besoin d'être copiée, à moins qu'elle ne soit évincée ou modifiée dans l'exécution sans protocole de Message Logging. La quantité de ce type de données est dépendant du type d'application, mais celles nécessitant une copie ne représentent souvent qu'une fraction des autres données. Par exemple pour une factorisation de Cholesky, il est possible d'écrire l'application en n'envoyant que les données finales du problème. Avec notre approche aucune copie de donnée spécifique au log ne sera faite, étant donné qu'elles ne seront jamais modifiées dans le reste de l'exécution.

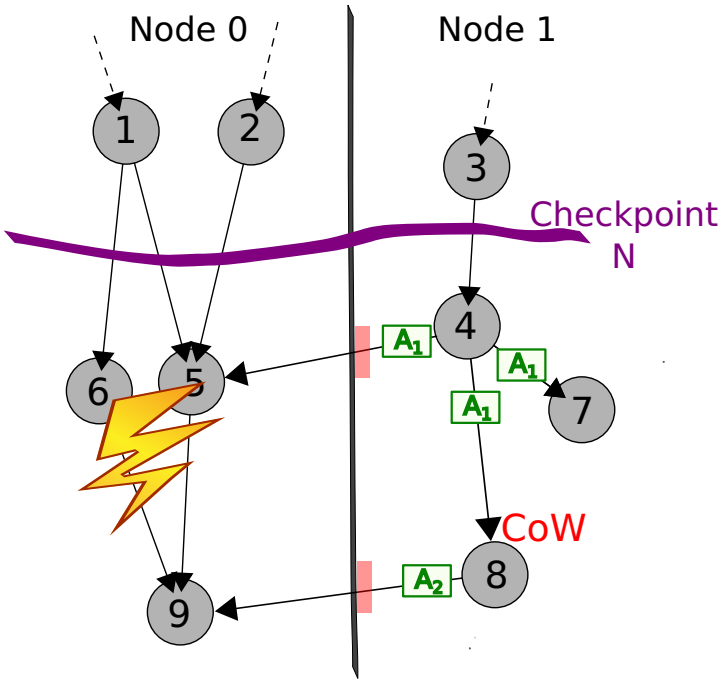
On peut aussi noter que les applications nécessitant une solution de tolérance aux pannes sont des applications qui arrivent à passer à une échelle où le taux de panne est problématique. Les applications qui sont limitées par les communications sont celles qui pourraient nous poser le plus de problèmes, pouvant demander une consommation mémoire excessive pour le log de message. Mais ces dernières applications ont justement des difficultés à passer à l'échelle à cause de ces communications [28]. Le ratio des gains par rapport à la puissance de calcul mise en œuvre est beaucoup moins intéressant que pour d'autres applications : il vaut alors mieux exécuter avec un nombre de machines moindre, ce qui induit un taux de pannes plus clément et pour lesquels on fera des checkpoints beaucoup moins souvent.

Au contraire, les applications passant bien à l'échelle communiquent relativement peu, la bande passante consommée n'est alors qu'une faible fraction de l'empreinte mémoire de l'application. Par exemple pour un *stencil*, seuls les bords d'un sous-domaine sont échangés, ce qui ne représente qu'une faible portion de ce dernier. Pour un code itératif comme un gradient conjugué, seul un vecteur est modifié et partagé entre les nœuds, la matrice qui occupe la plus grande partie de la mémoire n'est jamais modifiée ou partagée au cours de l'exécution. La taille du log de message ne devrait donc pas nous pénaliser.

Pour ces applications, la taille du log ne devrait pas être problématique. Mais d'autres applications créeront des logs à l'empreinte mémoire conséquente, et même dans ce cas ce ne sera pas un problème. En effet il possible d'indiquer à StarPU qu'une



(a) Cas favorable : la donnée A n'est pas modifiée, elle peut donc être simplement référencée dans le log de message sans avoir à effectuer une copie.



(b) Cas défavorable : la donnée A est modifiée par la tâche 8, elle doit donc être copiée (CoW, Copy on Write) pour que la version A_1 soit conservée dans le log de messages, pour pouvoir être réémise lors de la panne.

FIGURE 3.6 : Illustration des cas de mise en œuvre du Copy on Write pour le log de messages.

donnée n'est pas utile au calcul. C'est intéressant car une donnée devant être copiée pour le log est une donnée qui précisément n'est plus nécessaire au calcul. Si la capacité mémoire s'avère être limitée, StarPU copiera ce type de données en priorité sur le disque (swapping) pour libérer de l'espace.

Tout ceci nous fait bien voir qu'en dépit d'expériences réalisées, le coût en performance de notre approche sera bien plus acceptable que si l'on avait utilisé une solution de message logging existante.

En revanche les données devant être conservées pour le log de message peuvent à un certain point dans l'exécution ne plus être nécessaires. Le log de message étant intrinsèquement lié aux checkpoints, on peut en fait supprimer les logs de messages en utilisant le garbage collector des checkpoints. Il suffit simplement de supprimer le log de message de section N quand on supprime le checkpoint N.

3.2.8 Optimisation potentielle

Comme illustré sur la Figure 3.7, il se peut qu'après une panne, un nœud 1 ait toujours en mémoire une donnée reçue A produite par le nœud 0 après le dernier checkpoint, mais reçue avant que 0 ne tombe en panne. On peut faire en sorte que le nœud 1 renvoie A au nœud 0 avec des informations permettant d'identifier clairement quelle tâche a créé cette donnée. Ainsi quand le nœud 0 reçoit A après le redémarrage, et qu'il identifie sans ambiguïté quelle tâche l'a produite, il peut éviter l'exécution de cette tâche si A est la seule donnée écrite par la tâche. Si plusieurs données sont modifiées par la tâche, il faudra vérifier que toutes ces données sont présentes pour choisir de ne pas exécuter la tâche. Cela permet au nœud 0 de rattraper plus vite son retard dans le cadre d'un redémarrage local, en n'exécutant pas une tâche dont le résultat est déjà connu, sans pour autant qu'il ne soit sauvé dans les checkpoints. Si au cours de l'exécution 1 a supprimé A, elle ne sera pas envoyée à 0 et le fonctionnement serait le même que celui présenté dans les propositions précédentes. On a donc un mécanisme optionnel qui pourrait s'ajouter, sans devoir modifier l'implémentation qui va être détaillée dans la prochaine partie. Le seul pré-requis est un mécanisme de versionning des tâches afin d'identifier sans ambiguïté quelle tâche a produit une donnée. Il faudrait donc que chaque tâche soumise ait un ID unique mais identique pour tous les nœuds, ce qui n'est pas encore le cas avec StarPU. On peut noter que ce comportement s'apparente aux protocoles de vol de travail, et il sera intéressant d'intégrer cette optimisation quand StarPU-MPI permettra de faire ce type de ré-équilibrage de charge.

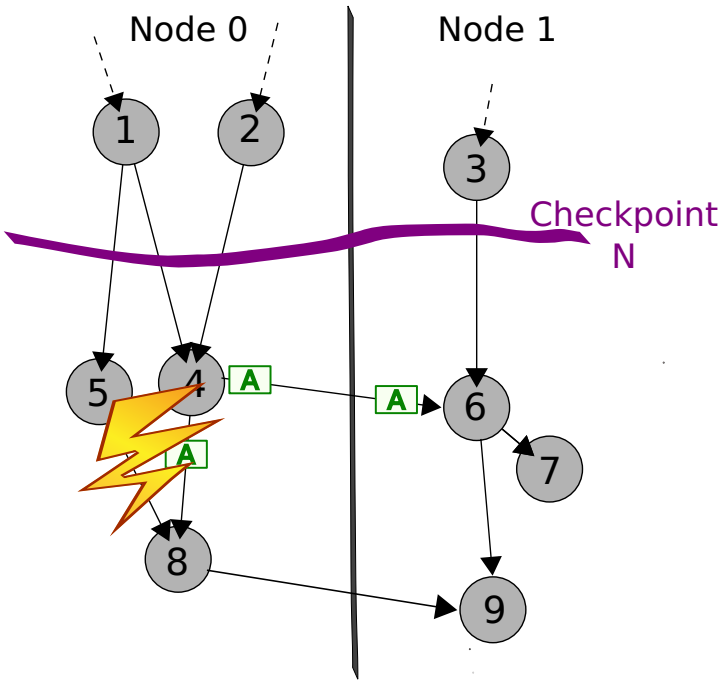


FIGURE 3.7 : Exemple d'optimisation éventuelle pour accélérer la reprise. Le nœud 1 pourrait être en mesure de renvoyer la donnée A au nœud 0, lui épargnant ainsi d'avoir à relancer la tâche 4.

3.3 Détails d'implémentation

Nous présentons dans cette section comment les propositions précédentes peuvent être implémentées dans StarPU. Une implémentation a été réalisée dans un premier temps dans un code maquette afin de valider que le fonctionnement de la solution proposée est conforme. Ce code maquette a été réalisé au début de la thèse et reprend les principes de gestion des communications de StarPU-MPI. StarPU est complexe et demande beaucoup de modifications si l'on veut implémenter l'ensemble des propositions, aussi il était très intéressant de valider dans un premier temps le fonctionnement de notre approche avant une intégration dans StarPU même. La maquette n'effectue en revanche pas de checkpoints, on se contente de redémarrer un nœud à l'état initial et d'effectuer le protocole de message logging. Cela nous a permis de valider que notre approche était concluante.

J'ai à la fin de ma thèse tenté d'implémenter le log de message et le redémarrage local dans StarPU-MPI, mais de nombreux choix d'implémentation se sont avérés inutilement complexes. L'implémentation s'est retrouvée trop compliquée, difficile à maintenir et donc instable. Par exemple j'ai d'abord cru qu'il était plus facile de compléter le log de message lorsque le test de la requête détachée retourne avec succès, pensant que l'implémentation serait plus simple ainsi ; mais de nombreux problèmes sont apparus. Le log de messages reçus nécessitant d'être complété à ce moment là, j'avais pris pour

acquis qu'il serait plus simple de factoriser le code en ayant une seule fonction pour compléter les logs de messages reçus et émis. Mais en choisissant de remplir le log ici, l'ordre dans lequel les requêtes sont insérées dans le log n'est pas celui dans lequel elles ont été soumises. J'avais élaboré un algorithme pour pouvoir conserver l'ordre dans le log de message, mais il s'est complexifié de plus en plus, rendant l'intégration incertaine. La prise de recul pendant la rédaction de ma thèse m'a permis de me rendre compte qu'il suffit juste de compléter le log de message au moment où la communication commence pour s'assurer que l'ordre est respecté. De manière générale, de nombreuses solutions plus simples que celles qui avaient commencé à être implémentées ont été trouvées pendant la rédaction. Aussi l'implémentation qui a été en partie réalisée demanderait à être recommencée. Nous présentons ainsi dans cette section ce qui doit être fait pour implémenter le log de message et la reprise locale dans StarPU, fort de l'expérience issue de l'implémentation dans le code maquette et de l'implémentation avortée dans StarPU.

Nous commençons par nous intéresser à l'implémentation du log de message, puis comment mettre en place la procédure de redémarrage local avec ULFM. Enfin nous reviendrons sur les modifications essentielles qui doivent être faites dans StarPU-MPI.

3.3.1 Implémentation du log de messages

Le log des messages n'est pas très compliqué à implémenter. On commencera par voir quelle modification doit être effectuée dans StarPU-MPI, avant de présenter l'implémentation des logs de messages respectivement en émission et en réception.

3.3.1.1 Modification des structures internes de StarPU-MPI

En 3.2.3, nous étions arrivés à la conclusion que pour pouvoir identifier correctement la section du log de message qui est nécessaire au redémarrage, il est possible de se baser sur l'ID du checkpoint avec lequel le checkpoint a redémarré : c'est notre horloge de Lamport. On avait vu que la section du log auquel appartient un message dépend de l'ID du dernier checkpoint soumis avant la soumission de la tâche qui a demandé la communication. Cette information n'est accessible qu'au moment de la soumission de la tâche, et elle doit être propagée jusqu'à ce que la communication soit soumise lors de l'exécution.

On doit donc veiller à ce que l'ID du dernier checkpoint soumis soit renseigné dans la structure des tâches soumises, et de passer cette information lorsque la tâche crée une requête d'envoi ou de réception lors de l'exécution. Il faut donc ajouter un champ `section` dans la structure des requêtes de communication, qui sont créées à la soumission et où l'information est accessible.

3.3.1.2 Le log de messages envoyés

L'objectif ici est clair : on veut garder une trace de tous les messages envoyés. Il faut entrer le message dans le log d'émission dès le début de la communication, donc au moment où la requête d'envoi est initiée et dépilée de la liste des requêtes prêtes. De cette manière on est certain que l'ordre des messages dans le log est le même que l'ordre de soumission des requête. Dès que la communication est postée, on peut considérer le message comme envoyé et le placer dans le log : la seule raison pour laquelle le message échouerait est que le nœud récepteur tombe en panne, auquel cas on ré-émettra la communication avec le log.

On choisi de remplir le log de message dès que l'envoi est initié. Le log de message peut être simplement implémenté comme n'importe quelle liste de requêtes. On crée donc une nouvelle requête qui correspond en tout point à celle soumise par StarPU pour envoyer les données. La référence vers les données doit en revanche être modifiée. À cet endroit les données sont indiquées avec un `starpu_data_handle_t`. Il ne faut pas utiliser le même handle car on ne veut pas que la donnée puisse être modifiée par StarPU. On peut utiliser la fonction `starpu_data_dup_ro()` pour dupliquer la donnée. Cette fonction permet de dupliquer le handle représentant cette donnée, en retournant un nouveau handle qui dispose de droit en lecture seule sur cette donnée. En pratique la copie de la donnée est systématique pour l'instant car l'implémentation n'est pas encore faite, mais il intégrera à terme un mécanisme de copy-on-write. Ceci permettra d'alléger le coût en mémoire du log de message, en évitant d'effectuer une copie spécifique pour le log de message tant qu'elle n'est pas modifiée. On pourra aussi marquer le handle avec la fonction `starpu_data_wont_use`, afin que StarPU pousse cette donnée sur disque en priorité au cas où le swap est nécessaire par manque de capacité mémoire.

3.3.1.3 Le log de messages reçus

Le log de message de message en réception est n'est pas plus compliqué à implémenter. Comme présenté en Section 3.2.4, ce que l'on souhaite in fine est de pouvoir compter le nombre de messages reçus pour chaque tag dans la section qui nous intéresse. L'ordre d'insertion dans le log n'est donc pas déterminant. On peut insérer une requête dans le log au moment où elle est considérée comme ayant réussi, contrairement au log de message en émission.

On utilise ici aussi une liste de requêtes StarPU-MPI pour stocker le log de messages. On crée une simple copie de la requête du message reçu. Il faut tout de même modifier le champ handle indiqué dans la requête. En effet on ne souhaite pas sauver le contenu du message ici, on peut se contenter de remplir le champ avec une valeur nulle.

On pourra noter qu'utiliser des listes de requêtes StarPU-MPI pour les logs de messages manque peut-être d'efficacité car uniquement quelques champs de la structure

requête ont vraiment besoin d'être sauvés dans le log. Aussi une structure plus adaptée permettrait de diminuer sensiblement l'empreinte mémoire du log de messages si nécessaire.

3.3.2 Implémentation du redémarrage local

Nous avons choisi d'utiliser ULFM (voir 1.2.4) pour plusieurs raisons. Tout d'abord la version distribuée de StarPU utilise MPI et nous souhaitons donc utiliser une solution qui y est associée. Bien qu'ULFM ne soit pas l'unique solution, le projet est très actif dans la communauté MPI. Le fait que cette solution soit ajoutée dans la future version 5.0 version de OpenMPI soutient également la pertinence de ce choix. Aussi, bien que le nombre de fonctionnalités offertes par ULFM soit limité, cela reste suffisant pour intégrer l'ensemble de nos propositions dans StarPU.

3.3.2.1 Détection de panne avec ULFM

ULFM propose un mécanisme de détection de panne. L'information de panne peut être divulguée à l'application de plusieurs façons, soit par retour d'erreur lors d'appels MPI, soit sous forme d'appel a une fonction spécifique avec `MPI_Comm_set_errhandler`. Si l'on choisit cette deuxième option, il faut savoir que l'on ne peut pas passer de paramètres personnalisés via cette fonction, le comportement est défini dans l'implémentation de MPI (OpenMPI passe uniquement une chaîne de caractère contenant le nom de la fonction qui retourne une erreur par exemple). Pour traiter correctement une panne, il faut avoir accès à certaines valeurs concernant l'échec, telles que le communicateur et la requête MPI qui a échoué : on n'aura pas accès à ces informations si l'on utilise une fonction `errhandler`.

Le retour d'erreur est donc notre seule option pour traiter les erreurs. Il faut par conséquent modifier le code de StarPU-MPI afin d'évaluer le retour d'erreur de chaque appel MPI. Cette tâche est lourde mais tout de même réalisable. L'intégration a été faite dans le code maquette qui reprend une partie allégée de StarPU, afin de valider l'approche sans avoir de trop nombreuses modifications nécessaires.

L'information de panne est donc remontée à l'application par valeur de retour lors d'un appel MPI. Mais une erreur de panne n'est pas toujours retournée lors d'un appel MPI, même si MPI a reçu l'information de la panne avant l'appel. En effet pour que l'appel MPI retourne une erreur, la panne doit influer sur le résultat de l'appel.

Par exemple si un nœud 0 a posté une requête `MPI_Irecv()` avec comme source le nœud 1, mais que 1 tombe en panne avant que la communication ne soit terminée, un appel à `MPI_Test()` sur cette requête retournera une erreur `MPIX_ERR_PROC_FAILED`. L'erreur est effectivement retournée car la panne remet en cause le résultat attendu par l'appel à `MPI_Test()`. En revanche si c'était le nœud 2 qui était tombé en panne,

le test de cette même requête ne retournera jamais l'erreur `MPIX_ERR_PROC_FAILED` car la panne ne remet pas en cause le résultat attendu, étant donné que la requête ne concerne que les nœuds 0 et 1. Si la source de la requête avait été `MPI_ANY_SOURCE` par contre, à ce moment-là une panne sur n'importe quel nœud peut influer sur le résultat attendu. L'appel à `MPI_Test()` retourne donc une erreur, mais qui est différente : `MPIX_ERR_PROC_FAILED_PENDING`.

3.3.2.2 La réactivité de détection

On verra que le traitement de la panne est un processus bloquant qui nécessite la contribution de l'ensemble des nœuds MPI. Le temps perdu lors du traitement est donc dépendant de la rapidité de la propagation de cette erreur. Les programmes qui progressent peu (ne faisant pas d'appel régulier à MPI et ne communicant pas avec tous les nœuds) peuvent donc mettre longtemps à récupérer l'information d'une panne. En pratique MPI peut avoir reçu en interne l'information de la panne, mais sans informer l'application si le résultat des appels à MPI n'est pas remis en cause par la panne. Dans StarPU-MPI, le thread de progression des communications soumet et teste en permanence une réception avec comme source `MPI_ANY_SOURCE`. Ceci est dû au fait que StarPU-MPI transmet les messages en deux temps avec d'abord une enveloppe contenant des métadonnées, puis un second message contenant les données (voir 1.3.3.2). Les réceptions de ces enveloppes sont faites avec une requête `MPI_Irecv()` en `MPI_ANY_SOURCE` qui est testée en permanence par le thread de progression. Le second message contenant les données est lui réceptionné avec une requête `MPI_Irecv()` qui précise le rang de la source.

Grâce à cette réception de `MPI_ANY_SOURCE` permanente, on est assuré que le runtime sera rapidement informé de la panne, et pourra réagir rapidement en conséquence.

Ces appels réguliers permettent aussi d'éviter un effet de bord du détecteur de panne proposé par ULFM, qui peut faussement déterminer qu'un nœud est en panne s'il ne fait pas régulièrement des appels à MPI [17].

3.3.2.3 Réparation du communicateur

Une panne qui se produit se traduit par le fait qu'un nœud MPI du communicateur n'est plus disponible. Si l'on souhaite redémarrer le programme il faut remplacer le nœud MPI en panne, et donc réparer le communicateur. En pratique un communicateur MPI ne peut pas être modifié, il faut en créer un nouveau. Les nœuds MPI restants doivent donc entrer dans une procédure commune afin de créer un nouveau communicateur remplaçant l'ancien, contenant cette fois le nœud qui remplace celui en panne. Le nœud entrant dans la procédure de réparation le fait par l'intermédiaire du thread de progression de StarPU-MPI. Ce thread est le seul de StarPU à faire des appels MPI, on

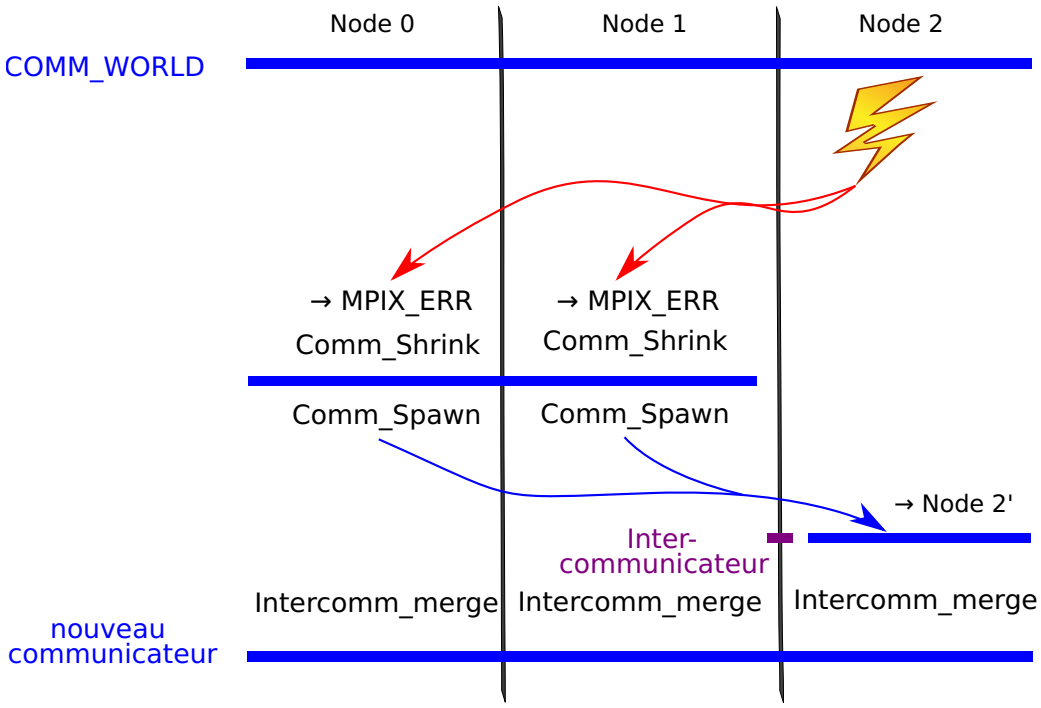


FIGURE 3.8 : Déroulement de la reconstruction du communicateur. Les nœuds survivants réduisent leur communicateur, puis lancent un nœud de remplacement, et les communicateurs sont rassemblés.

est donc certain qu'aucun appel MPI n'intervient pendant la réparation. La réparation est un processus bloquant car elle contient des collectives bloquantes, et donc les threads de progression des nœuds se retrouvent synchronisés. En revanche les threads workers de StarPU continuent d'exécuter les tâches en parallèle. Ainsi les nœuds continuent d'exécuter les tâches prêtes, tant que les données nécessaires sont disponibles. Donc même si la réparation est bloquante, elle ne bloque pas pour autant l'exécution des tâches et donc impacte relativement peu les performances. Elle empêche en revanche les communications entre les nœuds, donc chaque nœud ne pourra effectuer que les tâches pour lesquelles les dépendances locales sont disponibles.

La procédure de réparation, illustrée par la Figure 3.8 commence par réduire le communicateur original en un nouveau communicateur qui ne contient pas le ou les nœuds en panne en appelant `MPIX_Commm_shrink()`. Cette opération collective garantit une synchronisation entre les nœuds survivants à partir du moment où elle retourne. À noter que si une autre panne se produit pendant cette collective, cette dernière retournera sur les nœuds restants mais le communicateur contiendra tout de même ces nœuds en panne, mais ne retournera pas d'erreurs pour autant. Pour la suite de l'opération les erreurs doivent être vérifiées à chaque avancement, en intercalant des appels à la collective `MPI_Comm_agree()` pour s'assurer que les opérations provoquent le

résultat attendu sur l'ensemble des nœuds. Il faudra recommencer la procédure le cas échéant.

Après avoir réduit le communicateur au nombre de nœuds restants, on peut démarrer un nouveau nœud en utilisant la collective `MPI_Comm_spawn()`. C'est dans cette fonction que l'on donnera les options nécessaires au redémarrage, avec la structure `MPI_Info`. La clé `pernode` permet d'indiquer que l'on souhaite un seul nœud par machine. On peut aussi préciser une liste de machines sur lequel le nœud peut redémarrer avec la clé `hostfile`, notamment si le hostfile original ne dispose plus de machines libres. Le nœud redémarré va alors s'initialiser. Pour savoir s'il est un nœud rajouté après une panne, il appelle `MPI_Comm_get_parent()`, qui retourne le communicateur utilisé lors de `MPI_Comm_spawn()`. On peut discriminer le fait d'être un nœud redémarré ainsi, car si le nœud démarrait selon une exécution classique l'appel `MPI_Comm_get_parent()` n'aurait renvoyé aucun communicateur. Avant de recréer le nouveau communicateur qui sera celui réparé, le groupe de nœuds survivants doivent envoyer au nouveau nœud le rang du nœud en panne, qui lui sera attribué dans le nouveau communicateur. Ce rang peut être déterminé en comparant le communicateur original et le communicateur réduit, avec la fonction `MPI_Group_difference()`. Un seul nœud survivant doit évaluer et envoyer cette information au nouveau nœud. Enfin le nouveau communicateur est créé avec la collective `MPI_Intercomm_merge()`, qui rassemble le communicateur des survivants avec le communicateur du nœud redémarré. Les codes d'exemple proposés par les développeurs d'ULFM sont intéressants à étudier afin de mieux comprendre le fonctionnement du redémarrage [14].

Le nœud est maintenant redémarré, et un communicateur MPI réparé est utilisable. Il faudra utiliser ce communicateur pour tout nouvel appel à MPI. Par contre il est trop tôt pour supprimer l'ancien communicateur, celui-ci nous sera encore utile comme on pourra le voir en Section 3.3.2.7. En effet d'autres actions doivent être réalisées dans la procédure de réparation avant de reprendre l'exécution.

3.3.2.4 Annuler les communications avec le nœud redémarré

Les nœuds survivants ont probablement des communications en cours avec le nœud qui vient de tomber en panne. On itère donc sur la liste `detached_requests`, en filtrant pour ne garder que celles à destination du nœud redémarré. Il faut annuler ces communications pour libérer de l'espace mémoire, mais aussi pour mettre à jour les logs de messages.

On s'intéresse en premier aux requêtes d'envoi de message. Puisque l'on a choisi de remplir le log d'émission au début de la communication, la requête d'envoi peut être annulée sans qu'autre chose ne soit nécessaire, étant donné qu'elle est déjà dans le log.

Pour les requêtes de réception, on peut vérifier si le message a pu être reçu avant la

panne. En effet si on l'a reçu, on pourra débloquer des tâches sans attendre que le nœud redémarré ne la renvoie. Si le test d'une requête de réception s'avère être un succès, on peut la considérer comme accomplie et compléter le log des messages reçus. Si la communication n'a pas terminé avant la panne, elle retourne `MPIX_ERR_PROC_FAILED`, et on doit donc l'annuler. Il faudra resoumettre cette réception, donc on place la requête dans une liste temporaire, que l'on insèrera en tête de la liste `ready_recv_requests` une fois que l'on aura itéré sur tous les messages.

Il faut tout de même faire attention à ne pas tester la requête r qui a détecté la panne à l'origine, et qui a provoqué la procédure de réparation. En effet quand l'erreur `MPIX_ERR_PROC_FAILED` est retournée lors du test d'une requête, cette requête est marquée comme complétée par MPI. Ainsi lorsque l'on teste une nouvelle fois cette requête, elle ne retournera pas d'erreur mais `MPI_SUCCESS`. Si l'on re-teste la requête r ici, elle retournera donc `MPI_SUCCESS` alors que la donnée n'a pas été reçue. Il faut donc considérer que r est à resoumettre quelle que soit la valeur de retour de `MPI_Test()`.

3.3.2.5 Déterminer quel checkpoint doit être utilisé

On peut dès à présent déterminer quel était le dernier checkpoint complet avant la panne. Les nœuds ayant survécu exécutent une collective pour parvenir à un consensus. Les checkpoints étant indexés de manière croissante, le plus récent est celui dont l'ID est le plus élevé. Avec les messages de validation des checkpoints (comme vu en section 2.2.2), si le nœud k est en panne, on est sûr que son backup principal connaît quel est le dernier checkpoint local valide de k.

Néanmoins les autres nœuds doivent vérifier s'ils ont un risque de message en-transit comme évoqué en Section 3.1.

Chaque nœud commence par regarder quelles sont les réceptions depuis le nœud k qui ne sont pas encore terminées. Il faut donc regarder dans les listes des `ready_recv_requests` et `detached_requests`, mais aussi dans les requêtes que StarPU n'a pas encore soumis au thread de progression. Pour chaque requête trouvée, on regardera la valeur du champs `section` et on retiendra la valeur minimale. Si aucune requête n'est présente, on prendra l'ID du dernier checkpoint soumis. Le nœud $k+1$ devra en plus comparer cette valeur avec l'ID du dernier checkpoint local valide de k, et prendre le minimum. Si plusieurs nœuds sont en panne, on recommence autant de fois et on retiendra la valeur minimale. Un consensus est alors effectué en partageant la valeur obtenue avec un `MPI_Allreduce()` avec l'opération `MPI_MIN`. La valeur reçue est l'ID maximum du checkpoint qui pourra être utilisé pour redémarrer le ou les nœuds. Si plusieurs nœuds tombent en panne, on est obligé d'utiliser des checkpoints du même ID pour redémarrer les nœuds afin d'être sûr de ne pas avoir d'incohérence entre eux.

Pour le nœud en panne k (de rang le plus faible si plusieurs nœuds sont en panne), on

détermine quel checkpoint utiliser. On exécute donc `MPI_Allreduce()` avec l'opération `MPI_MAX` afin de déterminer quel est le dernier checkpoint utilisable. Les nœuds survivants indiquent alors l'ID de checkpoint le plus élevé avec lequel ils garantissent que l'on peut redémarrer le nœud k. Concrètement uniquement le (ou les si l'on a plusieurs templates) backup(s) principal(aux) vont mettre une donnée intéressante, les autres ne pourront garantir que l'on puisse démarrer plus loin que l'état initial et mettront 0. Chaque backup principal va donc mettre l'ID du dernier checkpoint local validé de k. Comme un nœud backup principal possède la liste des nœuds qui sont aussi backup dans le template, il sait si un de ces nœuds est en panne. Si un des nœuds backup est effectivement en panne, le nœud backup principal sait qu'il ne peut pas repartir plus loin que 0 avec son template, et mettra donc cette valeur. On recommencera autant de fois que l'on a de nœuds en panne et l'on retiendra l'ID de checkpoint le plus faible. Si l'ID final obtenu est 0, on ne peut pas continuer. On verra ce que l'on peut faire dans ce cas en Section 3.4. Si en revanche l'ID n'est pas nul, on est sûr de pouvoir continuer. Si, pour un des nœuds en panne, plusieurs templates sont utilisables, on utilisera celui de l'ID du template le plus faible (à ne pas confondre avec l'ID des checkpoints). Lors d'une réparation pour plusieurs pannes, il n'est pas nécessaire de démarrer plusieurs nœuds avec le même ID de template. On peut redémarrer des nœuds avec des templates différents, mais l'ID du checkpoint utilisé doit lui être identique.

Il faut noter tout de même qu'une fois que l'on a redémarré un nœud k à un checkpoint n, si une autre panne se produit sur un autre nœud p, on ne pourra pas redémarrer à un checkpoint antérieur à n. En effet, le log de message antérieur à n est perdu sur k car il n'est pas inclus dans le checkpoint. S'il n'y a pas d'interaction entre les nœuds k et p avant n on pourrait redémarrer éventuellement p avec un checkpoint plus ancien. Mais il est impossible de déterminer si c'est le cas. Cela peut poser problème si l'on a choisi de redémarrer le nœud k au checkpoint n car cela ne causait pas d'incohérence avec les autres nœuds, alors que le dernier checkpoint global cohérent était $n-2$ par exemple. Si lorsque l'on répare la panne de p, si on se rend compte que l'on ne peut redémarrer p qu'au checkpoint $n-1$, il est impossible de continuer. On peut faire un redémarrage global, mais on préfèrera redémarrer le nœud k au checkpoint $n-1$, même s'il n'est pas en panne, afin de rétablir une cohérence.

Après cela, chaque nœud pourra vérifier de manière autonome s'il a été chargé de sauver des données de ce checkpoint, et les envoyer au nœud redémarré si c'est le cas.

Le nœud qui est le backup principal(défini en Section 2.2.1) du nœud k pour le template que l'on a choisi d'utiliser, envoie au nœud redémarré les informations à propos du checkpoint qui va être utilisé. Ces informations contiennent l'ID du checkpoint utilisé mais aussi l'ID du template, ce qui est indispensable afin que le nœud redémarré interprète correctement le checkpoint reçu.

3.3.2.6 Partage du log de message en réception

On souhaite faire en sorte que le nœud redémarré n'envoie pas les messages qui ont été déjà reçus par les nœuds survivants. Les nœuds qui ont déjà reçu des messages doivent donc envoyer des informations auprès du nœud qui redémarre pour qu'il évite de lui-même les envois. Chaque nœud survivant détermine quelle est la section du log à utiliser, qui est en fait l'ID de checkpoint qui a été déterminé à l'étape précédente et qui sera utilisé pour redémarrer le nœud. Il compte ensuite combien de messages ont été reçus en provenance du nœud en panne pendant la dernière section du log. Le compte est relatif au tag du message reçu, et l'on aura autant de compteurs positifs que de tags différents reçus dans la section du log. Quand l'ensemble du log est parcouru, on peut communiquer au nœud redémarré les compteurs pour chaque tag. Le message sera envoyé comme un message de service en deux temps, avec une enveloppe pour indiquer le nombre de compteurs envoyés. L'envoi de l'enveloppe et des compteurs sont effectués comme un type spécifique de message service, ayant des tags réservés dans StarPU-MPI pour l'enveloppe et la donnée. Si le log de message est vide, on envoie tout de même une enveloppe précisant que le nombre de compteurs est 0.

De son côté le nœud redémarré place un flag indiquant qu'il doit respecter les logs de messages reçus des autres nœuds. Tant que ce flag est actif, le thread de progression veillera à ce que les envois soumis par StarPU au thread de progression ne soient pas initiés, tant que l'on n'a pas reçu les compteurs des nœuds que l'on cherche à joindre. Il doit ensuite poster autant de réception d'enveloppes que de nœuds ayant survécu à la panne précédente, afin de recevoir les compteurs.

À partir de là, il a fait tout ce qui était nécessaire dans la procédure de réparation, la suite des opérations concerne uniquement les nœuds survivants. Il peut donc quitter la procédure ce qui aura pour effet de débloquer l'appel à l'initialisation de StarPU-MPI effectué par l'application. L'application continuera alors normalement jusqu'à ce qu'il rencontre la définition du `checkpoint_template` qui correspond aux informations de redémarrage. À ce moment là StarPU comprendra qui est son backup pour chaque donnée du checkpoint, et peut donc poster les réceptions correspondantes. Lorsque l'application aura atteint la fonction `starpu_checkpoint_restore()`, StarPU attendra d'avoir reçu toutes les données du checkpoint. Quand ce sera le cas, il modifiera les données de l'application avec les valeurs du checkpoint.

Le nœud redémarré insère un checkpoint

Avant de rendre la main à l'application, on va insérer un nouveau checkpoint. Les premiers messages qui seront envoyés quand on dépilera le log seront en fait les checkpoints des autres nœuds. En effet quand on soumet le checkpoint dont l'ID est n, une nouvelle

section n du log de message est commencée, et on a choisi d'associer les communications dûes au checkpoints à la section n. Comme l'application positionne ses variables d'instrumentation afin de reprendre l'exécution à l'instruction qui suit le checkpoint, l'application n'exécutera pas la fonction d'insertion de checkpoint correspondant à celui utilisé au redémarrage. On choisit donc de l'exécuter ici, mais celui ci ne va pas incrémenter l'ID : si l'on utilise le checkpoint n pour redémarrer, on resoumet ici le checkpoint n. Vu que le checkpoint vient de recevoir les données pour redémarrer, il voudra sauver ce même checkpoint. Ce n'est pas grave car il n'a toujours pas le droit d'envoyer de messages à l'extérieur tant qu'il n'a pas reçu les compteurs, et les compteurs permettront en réalité de filtrer tous ces messages. Mais là où c'est intéressant, c'est qu'il recevra les données du checkpoint n de la part des nœuds dont il est le backup. Cela permet de reconstituer au plus vite des données de sauvegarde sur l'ensemble des nœuds, et ainsi ne pas s'exposer trop longtemps aux pannes. En effet les nœuds qui utilise le noeud redémarré comme backup ne pourront pas redémarrer s'ils tombent en panne, tant que le nœud redémarré n'aura pas reçu une nouvelle sauvegarde. À noter que si l'on n'avait pas voulu resoumettre un checkpoint ici, il aurait juste fallu associer les communications induites par le checkpoint n à la section $n-1$ (vu que les envois des checkpoints sont à la frontière des sections on peut choisir librement où les placer). Dans ce cas le log de message rejoué à partir de la section n n'aurait pas inclus le renvoi des checkpoints : dans ce cas il ne faut pas soumettre le checkpoint lors du redémarrage du nœud redémarré.

Quand les compteurs seront reçus de la part d'un nœud, on pourra recommencer à lui envoyer des messages. Il faudra faire attention à prendre en compte les compteurs, et chaque fois qu'un message a le même tag qu'un compteur, on ignorera l'envoi. On considèrera la communication comme achevée et on la mettra dans le log de message, en n'oubliant pas de décrémenter le compteur. Quand tous les compteurs auront atteint 0, on pourra baisser le flag et supprimer les compteurs.

Pour les nœuds survivants il reste des actions à effectuer pendant la procédure de redémarrage. Il faut maintenant choisir le comportement à adopter face aux communications en cours pendant la réparation de la panne. En effet des nœuds était en train de communiquer pendant la panne, et il faut s'assurer que ces communications aboutissent.

3.3.2.7 Utiliser `MPI_Comm_ack()` ou `MPI_Comm_revoke()` ?

Si l'on utilise `MPI_Comm_revoke()` sur l'ancien communicateur, on sait que tout les appels à `MPI_Test()` qui suivront sur ce communicateur retourneront une erreur `MPI_ERR_COMM_REVOKED`, quel que soit le nœud. On peut alors se baser sur le fait que cette erreur indique qu'une communication doit être rejouée. Néanmoins cela ne nous permet pas de garantir que le fonctionnement soit cohérent. On peut par exemple avoir

un message qui est initié avec un `MPI_Isend` sur le nœud A et reçu avec un `MPI_Irecv` sur le nœud B. Si une panne apparaît sur un autre nœud, et que la réparation se produit après que le nœud B ait fait un `MPI_Test` qui retourne avec succès sur la requête de cette communication, un problème peut apparaître. En effet si le nœud A n'a pas fait appel à `MPI_Test` avant la réparation, le futur appel après réparation retournera `MPI_ERR_COMM_REVOKED`. Cela est problématique car le nœud A est ainsi informé que l'envoi du message est un échec et pourrait donc en déduire que le nœud B ne l'a pas reçu, alors que celui l'a bel et bien reçu.

Une autre approche est de ne pas révoquer l'ancien communicateur et d'utiliser plutôt `MPI_Comm_ack()`. Cette fonction permet que les futures réceptions en `MPI_ANY_SOURCE` sur l'ancien communicateur ne retournent pas l'erreur `MPIX_ERR_PROC_FAILED_PENDING` pour les pannes qui ont déjà été traitées. On pourra prendre soin de vérifier que la liste des nœuds en panne qui ont été acquittés avec `MPI_Comm_ack()`, corresponde bien à la liste des nœuds que l'on vient de redémarrer, en utilisant `MPI_Comm_get_acked()`. Ainsi il est possible de continuer à tester les appels MPI soumis avant la panne sur l'ancien communicateur sans détecter la panne qui vient d'être traitée. Cela permet aussi de ne pas annuler inutilement les communications en cours qui peuvent se poursuivre pendant le traitement de la panne. Nous avons choisi cette approche car c'est la seule qui nous permet d'évaluer localement si une communication doit être rejouée ou non.

Utiliser `MPI_Comm_get_acked()` implique d'intégrer de nombreux mécanismes pour vérifier si une communication doit être rejouée, et j'ai passé plusieurs mois à les implémenter sans avoir pu terminer pour autant. Même si cette approche est intéressante étant donné que l'on n'interrompt pas l'ensemble des communications si une panne se produit, la difficulté d'implémentation n'est pas à négliger. On aurait par conséquent apprécié qu'ULFM propose un mécanisme permettant d'identifier clairement si une communication a pu être terminée sur l'ancien communicateur ou non, et ainsi déterminer sans ambiguïté si la communication doit être rejouée sur le nouveau communicateur. Une telle fonctionnalité ne correspond pas aux spécifications de `MPI_Comm_revoke()` et nécessiterait d'être définie séparément ; mais il faudrait tout de même voir si cette fonctionnalité est compatible avec les spécifications du standard MPI.

3.3.2.8 Pourquoi garder l'ancien communicateur ?

Comme dit dans la section précédente, le choix a été fait de conserver l'ancien communicateur, en prenant soin d'appeler `MPI_Comm_ack()` pendant la réparation. On utilisera le nouveau communicateur pour toute nouvelle soumission de communication, mais l'on garde l'ancien communicateur pour terminer les communications en cours. On rappelle qu'une communication pour StarPU-MPI s'effectue en deux temps : d'abord l'envoi d'un enveloppe contenant les métadonnées, puis un second message contenant les

données transférées. Dès qu'un nœud a initié l'envoi de l'enveloppe d'un message, il faut garantir que le transfert s'effectue même si une panne survient sur un autre nœud que les deux concernés. Cela impose qu'il faille garder une réception active d'enveloppe sur l'ancien communicateur, au cas où une enveloppe ait déjà été envoyée avant la panne.

Si un nœud reçoit une enveloppe sur l'ancien communicateur, c'est qu'elle a été émise avant la panne. Afin que le nœud récepteur sache quel communicateur utiliser lors de la réception des données, il faut garantir que l'émetteur utilise toujours le même communicateur entre l'enveloppe et le transfert de données. En pratique dans StarPU-MPI, l'envoi de l'enveloppe et des données se font séquentiellement avec des `MPI_Isend`, et une panne ne peut pas être détectée entre les deux envois, donc on est assuré d'utiliser le même communicateur pour l'émission des données. On doit donc veiller à utiliser le communicateur avec lequel on a reçu l'enveloppe pour recevoir les données.

Pour indiquer que le canal entre deux nœuds ne sera plus utilisé pour les prochaines enveloppes, chaque nœud doit envoyer une enveloppe de clôture vers tous les nœuds. Quand le thread de progression des communication aura repris après la panne, la réception de cette enveloppe de clôture garantit que l'on ne recevra plus de nouvelles enveloppes depuis ce nœud, car les enveloppes utilisent un tag unique. Une fois que l'on a reçu l'enveloppe de clôture de la part de l'ensemble des nœuds qui sont encore vivants dans l'ancien communicateur, on peut envisager de libérer le communicateur avec `MPI_Comm_free`. Il faudra toutefois attendre que l'ensemble des requêtes initiées sur cet ancien communicateur soient terminées avant d'effectuer cet appel. On se basera donc sur le nombre de requêtes initiées sur le communicateur (i.e. dans la liste de requêtes détachées), puis on exécutera l'appel à `MPI_Comm_free` quand ce compteur atteindra 0.

3.3.2.9 Soumettre les envois des données du checkpoint de redémarrage

La prochaine étape dans la procédure de réparation est de préparer le nœud redémarré à la reprise d'exécution, comme illustré à la Figure 3.9. C'est à ce moment qu'on lui envoie le contenu du checkpoint. On a pu déterminer en consensus quel checkpoint doit être utilisé, et chaque nœud vérifie s'il a sauvé des données de ce checkpoint, et les envoie au nœud redémarré si c'est le cas.

Les envois de messages de checkpoint sont effectués avec le même système de communications que les données de calcul. On soumet donc tous les envois de données de checkpoint comme si c'était des données de calcul, en les insérant directement dans la liste des requête prêtes. Le transfert du checkpoint est prioritaire car l'on souhaite que le nœud redémarré commence a travailler au plus vite afin de rattraper le retard. Pour forcer la priorité face aux communications avec les autres nœuds, on insère les requêtes d'envoi de données du checkpoint en tête de la liste. Cela permet aussi de bien

séquencer la réception, à savoir que le nœud reçoive les données de checkpoint avant les données de calcul. Si les priorités pour les envois sont activées, on peut attribuer a ces transferts la priorité maximale.

3.3.2.10 Envoi du log de message

Une fois le contenu du checkpoint envoyé, les messages du log doivent être envoyés. Ceux-ci doivent être envoyés comme des messages normaux, car ils sont attendus comme tels par le nœud redémarré. On doit donc les insérer dans la liste des requêtes prêtes. Il doivent plus précisément être insérés entre l'envoi des données de checkpoint et les requêtes qui n'ont pu être initialisés avant la panne. En pratique on insère d'abord en tête de liste les requêtes d'envoi du log de message, puis de nouveau en tête de liste les requêtes d'envoi des checkpoints. De cette manière on sait que l'ordre d'envoi sera bien en premier lieu les checkpoints, puis le log de message, et enfin les communications de l'exécution normale.

À ce moment-là aucun message n'est encore envoyé, on a juste mis en place les requêtes pour que le thread de progression envoie les données dans le bon ordre et de manière asynchrone.

3.3.3 Bilan de la procédure de redémarrage

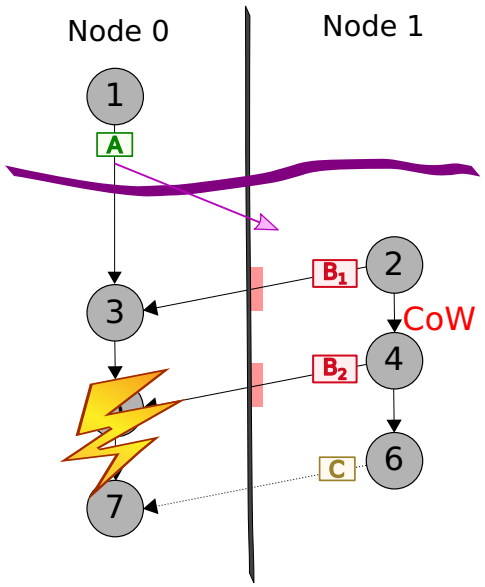
On peut revoir brièvement les différentes étapes nécessaires au redémarrage. Cette fois on distinguera la procédure qui doit être appliquée par les nœuds survivants à une panne de celle qui est effectuée en parallèle par le nœud redémarré.

Les nœuds survivants

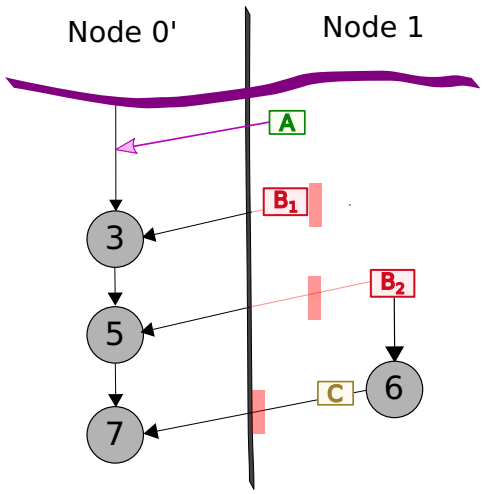
Il est important de prendre en considération qu'une panne peut survenir pendant cette procédure. Il faut donc vérifier les appels MPI qui peuvent retourner un erreur, et vérifier que tous les nœuds ont réussi en appelant la collective `MPI_Comm_agree()`. Si une panne se produit, les nœuds survivants doivent recommencer la procédure.

La réparation est initiée par la détection de la panne, c'est-à-dire quand une erreur `MPIX_ERR_PROC_FAILED` ou `MPIX_ERR_PROC_FAILED_PENDING` est remontée à StarPU-MPI. Ici tout est réalisé par le thread de progression des communications de StarPU-MPI. La procédure est bloquante, on sait qu'il n'y a pas d'appels MPI réalisé par d'autres threads, le thread de progression étant le seul à faire ces appels.

1. La première phase consiste à réparer le communicateur. Les collectives à l'intérieur provoquent le comportement bloquant de la réparation. Un nouveau nœud MPI est démarré et entre dans sa phase d'initialisation, puis un nouveau communicateur est créé où le nouveau nœud prend le rang de celui en panne.



(a) Le nœud 0 a envoyé au nœud 1 la donnée A contenue dans son checkpoint local (flèche violette). Le nœud 1 envoie des données B_1 et B_2 utiles pour les tâches du nœud 0, ces messages sont ajoutés au log de messages (rectangles roses). La tâche 6 n'est pas encore finie, la donnée C n'est donc pas encore envoyée. Le nœud 0 tombe alors en panne.



(b) Le nœud 0' a redémarré. Le nœud 1 s'en rend compte alors qu'il n'avait pas encore fini la tâche 6. Il doit d'abord envoyer au nœud 0' le contenu du checkpoint local, donc la donnée A. Il doit ensuite lui envoyer le log de messages, donc les données B_1 (dont il avait dû faire une copie) et B_2 (dont il n'a pas eu besoin de faire de copie). Le nœud 0 peut alors redémarrer et reçoit normalement les messages venant du log. La données C sera envoyée normalement (et ajoutée au log de messages).

FIGURE 3.9 : Exemple de reprise locale avec checkpoint et log de message.

2. La seconde opération s'assure que chaque nœud survivant annule les communications avec le nœud qui est tombé en panne, tout en veillant à correctement mettre le jour les logs de messages.
3. Durant la troisième phase, les nœuds survivants déterminent en consensus quel est l'ID du checkpoint qui doit être utilisé pour redémarrer le nœud. Le nœud backup principal du checkpoint utilisé envoie au nœud redémarré les informations lui permettant de savoir quel checkpoint va être utilisé. Le consensus est la dernière collective réalisée dans la procédure, à partir de là les nœuds survivants évoluent en parallèle de manière asynchrone.
4. Ensuite chaque nœud survivant va analyser le log des messages reçus afin d'établir des compteurs de messages reçus, pour chaque tag. Après cela ils envoient au nœud redémarré chaque compteur et le tag associé.
5. La phase suivante prépare la clôture de l'ancien communicateur, tout en permettant de finaliser avec succès les communications initiées avant la panne. Les nœuds survivants envoient une enveloppe de clôture à tous les autres nœuds survivants sur l'ancien communicateur et font ensuite appel à `MPI_Comm_ack()`.
6. Enfin on configure le thread de progression afin qu'il renvoie les données de checkpoints au nœud redémarré (s'il y en a) puis le log de message avant toutes les autres requêtes déjà postées..

La procédure de réparation est terminée et le thread de progression peut retourner là où il en était. Comme on a pris soin de gérer le comportement à adopter en manipulant les listes des requêtes prêtes, le thread de progression va effectuer les opération de manière autonome et l'on peut donc continuer à travailler normalement. La seule tâche à faire en comparaison d'une exécution sans panne est de vérifier quand l'ancien communicateur peut être fermé. Pour cela on attend de recevoir une enveloppe de clôture de l'ensemble des nœuds survivants, et quand cela arrive on peut marquer le communicateur comme à supprimer. S'il n'y a pas de requêtes en cours (compteur de requêtes détachées) sur ce communicateur, on peut le fermer dès maintenant. Sinon on attendra que la dernière requête termine avant de le fermer.

Pour le nouveau nœud

Le nœud redémarré commence son exécution comme un exécution normale. C'est dans la fonction d'initialisation de StarPU-MPI que l'on vérifie si l'on est dans le cas d'un redémarrage après panne, après le démarrage du thread de progression des communications. Les opérations MPI doivent être réalisées par ce thread de progression, donc on lui soumet la demande de procédure. L'appel à la fonction d'initialisation reste bloqué tant que le thread de progression n'a pas indiqué que la procédure de réparation

est terminée. Le thread de progression va de son côté se rendre compte qu'une procédure a été demandée et va donc la lancer.

1. Le nœud redémarré attend qu'un des nœuds survivants lui indique le rang qu'il va prendre dans le nouveau communicateur, puis le nouveau communicateur est créé.
2. Il attend ensuite de recevoir les informations sur le checkpoint qui va être utilisé pour redémarrer.
3. Il poste ensuite autant de réceptions qu'il y a de nœuds, afin de recevoir les compteurs des messages déjà reçus par les nœuds. Puis on place le flag qui imposera au thread de progression de vérifier s'il peut envoyer les données ou non, selon si l'on a reçu les compteurs.

La procédure de réparation est à présent terminée, et l'on peut débloquer le thread de soumission. L'application continue son exécution normale jusqu'à ce qu'elle rencontre la définition du checkpoint template qui lui permettra d'interpréter les données de checkpoint reçues. Puis l'application sera bloquée sur l'appel `starpu_checkpoint_restore()` tant que les données du checkpoint ne seront pas correctement reçues et mises en place. Une fois cela réalisé, l'application reprend son exécution normale.

3.3.4 Implémentations supplémentaires

Les précédentes descriptions n'incluaient pas ce qui va être présenté ici, pour ne pas surcharger cette partie qui était déjà bien complexe. Ce qui suit n'est pas pour autant optionnel, et doit être également implémenté afin d'avoir une solution complète.

3.3.4.1 Maintenir la cohérence de cache de StarPU

Le comportement à adopter pour maintenir la cohérence du cache doit se faire à plusieurs niveaux, afin de résoudre le problème abordé en Section 3.2.5. Il faut modifier le cache de communication de StarPU, pour ne pas juste indiquer une valeur booléenne "envoyée", mais aussi la section à laquelle a été effectivement envoyée le message. On lui ajoute un champs `section`. Lors de la soumission d'une tâche à la section n, si la valeur du cache est "non-envoyée", c'est qu'elle n'a jamais été envoyée et on l'enverra normalement, en mettant à jour le champs `section` à la valeur n. À la section $n + i$, si on soumet une tâche qui veut renvoyer cette donnée, la valeur du champs `section` du cache sera n. Dans ce cas on créera une requête `FAKE_SEND`, qui aura dans le champs `section` la valeur $n + i$, mais aussi un champs `origin_section` qui aura la valeur n. Cette requête sera traitée par le thread de progression des communications comme une requête à insérer dans le log de message mais sans l'envoyer.

Quand on devra renvoyer le log de message, il faudra filtrer ces requêtes `FAKE_SEND`. En effet on avait dit en Section 3.2.5 que l'on prend en compte ces requête uniquement si la requête qui a effectivement reçu la donnée est causalement antérieure au checkpoint. Cela veut dire que l'on n'envoie cette requête que si la valeur de son champ `origin_section` est inférieure à la section du log que l'on cherche à renvoyer. À noter qu'il y aura des requêtes `FAKE_SEND` en plusieurs exemplaires pour un même tag. Dans la section du log sélectionnée on prendra en compte la requête la plus ancienne, car il ne faut poster ces requêtes qu'une seule fois par tag.

Il faut faire exactement la même chose pour les données reçues : modifier le fonctionnement du cache, créer des requêtes `FAKE_RECV`... Et les conditions seront les même sauf que cette fois on incrémentera le compteur associé au tag de la requête plutôt que de resoumettre la requête, conformément à notre approche du log de messages reçus (Section 3.2.4).

3.3.4.2 Tester les appels MPI

Il faut tester la valeur de retour de chaque appel à MPI qui est susceptible d'échouer à cause d'une panne, afin de détecter les pannes. Ces appels sont nombreux dans le code de StarPU-MPI, aussi la modification peut être lourde. Également il faudra tester tout nouvel appel MPI qui pourra être ajouté dans le futur. Définir des macros qui donne le comportement qui teste la valeur de retour correctement selon si l'on a activé ou non la tolérance aux panne à la compilation de StarPU peut être une solution qui limite la complexité de maintenance.

3.3.4.3 Choisir le bon communicateur à utiliser

Quand une panne arrive on ne peut pas réparer directement le communicateur MPI, on doit en créer un nouveau. StarPU-MPI expose les communicateurs à l'application, mais on ne veut pas pour autant que l'application soit exposée au changement de communicateur après une panne. De plus le cache de StarPU-MPI ne doit pas non plus être exposé à ce changement. Plus concrètement on veut garantir que l'on puisse continuer à utiliser la valeur `MPI_COMM_WORLD` dans l'application et dans StarPU-MPI avant et après une panne. Il faut donc un mécanisme qui n'expose le changement de communicateur qu'aux commandes concernées. En pratique, uniquement les appels MPI effectués par le thread de progression lui-même ont besoin d'être exposés au changement de communicateur.

On propose donc un mécanisme d'abstraction du communicateur qui permet une correspondance entre le communicateur vu par l'application et StarPU (que l'on nomme communicateur application) et le communicateur à utiliser pour les appels MPI (que l'on nomme communicateur MPI). Il faut donc avoir une fonction qui retourne le

communicateur MPI que l'on doit utiliser en fonction du communicateur application. En pratique si l'on demande avant une panne le communicateur MPI à utiliser pour `MPI_COMM_WORLD`, la fonction retourne ce même communicateur, alors que si l'on appelle cette fonction après une réparation elle retournera le nouveau communicateur MPI contenant le nœud remplaçant du nœud en panne. Le seul endroit où l'on ne doit pas utiliser cette fonction est lorsque l'on poste la réception d'une donnée suite à la réception d'une enveloppe. Vu qu'une donnée est toujours envoyée sur le même communicateur MPI que l'enveloppe associée, on doit utiliser le communicateur MPI sur lequel l'enveloppe a été reçue. L'implémentation de ce mécanisme a été partiellement réalisée dans la branche `starpu_abstract_comm`.

3.4 Améliorer la couverture de pannes

On a pu voir que si l'on est sûr de pouvoir tolérer une panne d'un nœud isolé, il peut y avoir des cas de pannes multiples qui ne sont pas couverts. Il est possible de moduler la probabilité que cela se produise en manipulant les schémas de réplication.

3.4.1 Diminuer le taux de panne irrécupérable

Dans notre approche, nous avons laissé l'opportunité à l'utilisateur de choisir plusieurs nœuds pour sauver les différentes données du checkpoint. S'il est rentable de faire cela pour diminuer le coût des checkpoints en bénéficiant au mieux de la réplication naturelle des données par l'application, cela augmente en revanche la sensibilité aux pannes multiples. En effet, si un nœud backup tombe en panne en même temps que le nœud d'origine du checkpoint, on ne pourra pas reprendre l'exécution. Le nombre n de nœuds backup qui contribuent à un même checkpoint va donc directement augmenter la probabilité d'avoir des pannes irrécupérables. Cette probabilité est certes bien plus acceptable que le taux de panne du super-calculateur, mais il faudra veiller à ne pas choisir un nombre n de nœuds de backup trop grand pour un même checkpoint. Il faudra choisir un nombre n le plus petit possible, tant que l'on peut bénéficier suffisamment de la réplication effectuée par l'application. Il est possible de compenser cela en utilisant plusieurs stratégies de sauvegarde comme présenté en Section 2.2.1.

En utilisant deux templates lorsque l'on insère un checkpoint, chacune des données d'un checkpoint sera répliquée sur deux nœuds, tels que défini dans chacun des templates. Cela diminue énormément la probabilité d'avoir une panne irrécupérable. Si la seconde stratégie de sauvegarde permet aussi de bénéficier de la réplication de données induite par le calcul, il sera très efficace d'effectuer une duplication des checkpoints de cette manière.

Ce qui est intéressant et qui émerge de cela, c'est que l'on peut moduler le coût

de sauvegarde en fonction du taux de panne que l'on considère acceptable. Selon les applications il sera peut-être plus efficace de réduire le nombre de nœuds backup par template, quitte à augmenter le coût du checkpoint, afin d'avoir de meilleures chances de réussir ; ou bien il sera parfois plus efficace de dupliquer les checkpoints si cela n'augmente pas trop le coût de la sauvegarde.

On peut diminuer autant que l'on veut le taux de panne irrécupérable en utilisant des stratégies de récupération différentes, elle n'atteindra jamais zéro. Et lorsqu'une telle situation se présente, notre approche n'a pas d'autre choix que d'abandonner l'exécution. Mais il est possible de compléter ce que l'on a fait si l'on considère que le taux de panne obtenu n'est pas acceptable en comparaison du coût en performances.

3.4.2 Compléter la solution avec du stockage stable

Lorsque pour une application, on ne peut pas trouver de stratégie de réplication permettant d'avoir un taux de pannes irrécupérables acceptable sans que les performances n'en pâtissent, les checkpoints multi-niveaux sont une alternative très intéressante.

On pourrait commencer par choisir au départ si l'on veut activer la sauvegarde des checkpoints sur les autres nœuds ou pas. En effet si notre approche ne peut pas suffisamment bénéficier de la réplication induite par le calcul, on peut vouloir la désactiver ; si l'on doit envoyer les données sur un stockage stable, il faudra les transmettre deux fois (une fois vers un nœud backup et une fois vers le PFS) et la bande passante consommée par les checkpoints sera doublée.

On peut ensuite stocker les données de checkpoint sur le disque local à une certaine fréquence, afin de s'assurer que les sauvegardes puissent être accessibles si la panne ne s'avère être qu'un crash de l'application et que les données sont donc récupérables. On peut dans un second temps sauver à une fréquence moindre sur le stockage stable comme le PFS. Ces approches sont les mêmes que celles effectuées par les solutions multi-niveaux telles que FTI ou VeloC. Des travaux réalisés [48, 63] nous permettent de penser qu'il sera possible de pouvoir continuer à effectuer les checkpoints de manière incrémentale et différentielle sur les différents supports. Avec une interface adaptée on pourra faire en sorte que les données de checkpoint sur ces supports puissent être chargées par StarPU facilement, permettant de continuer à faire du rollback local. Il faudra néanmoins que le garbage collector du message logging n'efface pas des données qui sont nécessaires pour redémarrer avec un checkpoint sur le PFS, qui peut être relativement ancien et imposer de garder de nombreuses sections du log de message. On pourra alors choisir entre s'exposer au risque de devoir faire un redémarrage global et garder une grande portion du log de messages en mémoire.

Conclusion

À mesure que la puissance des super-calculateurs augmente, leur taux de pannes ne fait que croître. La capacité de continuer le calcul malgré une panne devient déterminante, alors que le runtime StarPU ne proposait jusqu'ici aucune solution. Maintenant que l'occurrence de ces phénomènes ne peut plus être négligée, il était nécessaire de proposer une solution de tolérance aux pannes pour StarPU. Si une panne peut survenir à cause de phénomènes aux sources multiples, elle se manifeste toujours de la même façon : une communication est interrompue. Cela peut être diagnostiqué pendant le calcul, et l'on peut alors agir en conséquence. Après avoir cherché dans l'état de l'art les différentes méthodes qui permettent de continuer un calcul malgré tout, aucune ne s'est avérée parfaitement adaptée à notre cas. Et quand elles s'avéraient compatibles, nous avions vu qu'il était possible de mieux mettre à contribution les spécificités du runtime et ainsi obtenir de meilleures performances. Les recherches ont été menées avec une motivation principale : par exemple durant une factorisation de Cholesky, si un nœud de calcul est perdu, on peut reprendre le nœud perdu en utilisant uniquement des données déjà disponibles sur d'autres nœuds pour les besoins applicatifs, ce qui se traduit par le fait que l'on n'a besoin d'aucun transfert supplémentaire pour pouvoir sauver les données du calcul.

Plusieurs questions sont soulevées par cette remarque. Le runtime a-t-il besoin d'informations supplémentaires pour savoir comment reprendre l'exécution d'un nœud ? Si oui quelle est la quantité d'information à lui fournir ? Comment les exprimer au runtime ? On sait reprendre si un seul nœud tombe en panne, mais peut-on extrapoler à la situation où plusieurs nœuds tombent en panne en même temps ? Sinon, qu'est ce que l'on doit exprimer de plus au runtime ? Toutes ces interrogations nous ont amené à proposer trois contributions, énumérées ici et résumées plus bas :

— Une solution de checkpoint de niveau application adaptée à StarPU avec une interface "standard", qui effectue les sauvegardes de manière asynchrone et diluée au cours de l'exécution, avec un checkpoint incrémental intégré avec le cache applicatif ;

— Une solution de message logging optimisée pour StarPU, permettant de pouvoir opérer un redémarrage local, i.e., remplacer uniquement les nœuds en panne

tandis que les autres continuent leur calcul ;
— Une stratégie de sauvegarde des checkpoints dans la mémoire des autres nœuds, qui permet de bénéficier de la réplication de données déjà présente dans l'application en utilisant davantage le cache applicatif.

Contributions

Les checkpoints de niveau application sont une solution qui offre un bon équilibre pour StarPU. Si l'on prend le temps de regarder le fonctionnement d'une application utilisant StarPU, on se rend compte qu'il comprend de nombreux phénomènes qui sont de mauvais augure d'après l'état de l'art. En effet le comportement de StarPU n'est pas déterministe en lui même : d'une exécution à l'autre, les tâches parallèles ne sont pas exécutées dans le même ordre, les données ne sont pas envoyées dans le même ordre... on pourrait croire que l'on est contraint d'utiliser un protocole de tolérance aux pannes lourd, mais il n'en est rien. Le runtime s'occupe juste d'exécuter les tâches soumises par l'application, et son état interne n'est pas critique en lui-même ; on n'a donc besoin de sauver aucune de ses propres données. En effet, du point de vue de la sémantique de l'application, StarPU est en fait déterministe. On peut en réalité insérer des checkpoints dans la soumission de l'application, de la même manière que l'on le réalise avec les bibliothèques FTI ou VeloC dans d'autres applications, car grâce au modèle STF, l'application s'écrit de manière séquentielle. On peut ainsi générer des checkpoints locaux qui appartiennent tous à un checkpoint global cohérent, sans besoin de coordination physique car la soumission apporte une coordination logique. Nos checkpoints contiennent exactement les même données que si l'on avait placé des checkpoints avec les bibliothèques citées ; mais ici on pourra initier les sauvegardes donnée par donnée tout en continuant à exécuter des tâches indépendantes, vu que l'on sait quelle donnée doit être sauvegardée et quand la donnée est prête. De plus le runtime suit les modifications de données et il sait quelles données ont été modifiées depuis le début du calcul ou depuis le dernier checkpoint ; on peut ainsi réaliser des checkpoints différentiels sans aucun calcul supplémentaire. En revanche nous ne pouvions pas nous inspirer des solutions développées pour Parsec par exemple, car StarPU ne peut pas accéder à sa guise aux différentes portions du graphe de tâches, notamment aux portions exécutées ou non encore soumises. Le code de soumission est la seule entrée pour pouvoir reprendre une partie du graphe déjà soumis. Comme ce code de soumission est effectué par l'utilisateur, StarPU ne peut pas faire le choix d'exécuter arbitrairement certaines portions du code. Laisser l'utilisateur insérer les checkpoints est donc nécessaire ; mais étant donné que notre interface est très proche des autres solutions existantes, l'effort demandé n'est pas un écueil.

Le message-logging est une fonctionnalité qui nous permet d'opérer un éventuel

redémarrage local. En effet les checkpoints sont suffisants si l'on souhaite redémarrer l'ensemble des nœuds ; mais si l'on souhaite redémarrer uniquement certains nœuds, il faut pouvoir relancer des communications déjà émises qui sont attendues par les nœuds redémarrés. Le log de message est une solution efficace pour palier ce problème, et nous avons pu déterminer comment l'implémenter à l'aide d'ULFM. Avec StarPU on a pu effectuer des optimisations significatives par rapport aux approches classiques, tout en induisant une latence totalement négligeable.

Les deux contributions précédentes étaient nécessaires pour atteindre la troisième contribution, qui consiste à sauver les checkpoints dans la mémoire d'autres nœuds. Les checkpoints, le message logging et le redémarrage local ont été pensés avant tout pour atteindre cet objectif là, mais il sera facile d'adapter ces contributions pour sauver les checkpoints sur d'autres supports et pour effectuer un redémarrage global. Sauver les checkpoints sur d'autres nœuds n'a du sens que si l'on peut exprimer des checkpoints de taille relativement faible. On a pu voir que l'overhead est complètement dépendant de la taille des checkpoints, mais qu'il est possible d'atteindre une overhead léger si l'on obtient des checkpoints de taille faible, voire un overhead nul lorsque l'on paramètre les sauvegardes de manière fine pour une décomposition de Cholesky par exemple. Probablement diverses applications ne peuvent exprimer des checkpoints de taille faible sans perdre en efficacité, mais certaines classes d'applications le pourront et il sera dans ce cas intéressant d'utiliser cette stratégie de réplication.

Perspectives

L'interface des checkpoints proposée est un élément fondateur pour la suite des implémentations de tolérance aux pannes qui seront réalisées dans StarPU. La capacité d'exprimer ainsi des checkpoints dans une exécution asynchrone est particulièrement notable. On pourra facilement utiliser cette partie pour s'ouvrir à des checkpoints multi-level, afin de palier au fait que la stratégie de sauvegarde sur les autres nœuds n'est pas adaptée à certaines applications. Le redémarrage global n'a pas été étudié en détail dans cette thèse, mais il serait intéressant de le faire rapidement car l'on a cherché à proposer une solution efficace pour le cas particulier d'une seule panne. Il faudra s'intéresser à savoir comment stocker des checkpoints incrémentaux de manière efficace, tout en laissant la possibilité de supprimer les données des anciens checkpoints qui ne servent plus aux nouveaux checkpoints. Il sera intéressant d'adapter notre approche avec des checkpoints composites, où l'on a différentes portions de code qui sont protégées avec différentes stratégies. Cette façon de faire est compatible avec ce qui a été réalisé avec seulement quelques modifications, et permettrait en plus de protéger les sous-fonctions de soumission à une granularité plus fine, de manière transparente à la fonction qui l'appelle. J'ai implémenté une première version des travaux présentés ici, mais beaucoup

de travail reste à accomplir. L'implémentation continue néanmoins, notamment car le projet européen MicroCard [9] compte parmi ses objectifs d'utiliser StarPU avec le modèle de tolérance aux pannes proposé.

Par la suite on pourra chercher des applications qui peuvent potentiellement bénéficier de la stratégie de réplication sur les autres nœuds en exprimant différemment l'ordre de soumission. Des compilateurs source-to-source avec une analyse polyhédrale peuvent être de bons outils pour transformer l'ordre de soumission. Il sera également intéressant d'utiliser des outils d'analyse statique afin de désigner comme backup les nœuds qui ont de toutes façons besoin de la donnée afin de minimiser le coût du checkpoint, et ainsi décharger l'utilisateur de cette tâche. De plus, choisir de séparer un checkpoint sur plusieurs nœuds augmente le risque de panne irrécupérable, mais dupliquer les checkpoints réduit ce risque : on pourra également utiliser des outils d'analyse pour chercher la meilleure stratégie de réplication à adopter, dans le but d'atteindre un taux de panne irrécupérable acceptable tout en minimisant l'overhead dû aux checkpoints. On se demande aussi s'il n'est pas possible d'utiliser des compilateurs source-to-source pour insérer automatiquement les checkpoints, en respectant au mieux les critères qui sont partiellement identifiés pour avoir une bonne efficacité, tout en automatisant l'instrumentation de code nécessaire au redémarrage. Les expériences réalisées ont entre autres mis en avant que les priorités affectées au checkpoint avaient un impact sur les performances : une priorité de checkpoint trop haute ralentit potentiellement le calcul, alors qu'une trop faible retarde potentiellement l'instant où le checkpoint est validé, ce qui nous expose à perdre davantage de travail en cas de panne. Diminuer les priorités des checkpoints est un moyen de diminuer l'overhead des checkpoints, mais qui en contrepartie augmente potentiellement le temps perdu en cas de panne ; on remarque un parallèle avec le fait que diminuer le nombre de checkpoints augmente le temps perdu si une panne survient, et il serait intéressant d'étudier si la formule de Young/Daly permet de trouver la priorité de checkpoint optimale [33].

On peut aussi noter qu'il sera intéressant de récupérer les signaux lancées par l'OS en cas d'erreur détectée mais non réparable, et tenter de ré-exécuter la tâche qui a subi cette erreur. Pour l'instant on laisse ces signaux générer une panne, mais il sera intéressant de traiter l'erreur à plus bas niveau afin de perdre moins d'avancement dans le calcul. On pourra aussi chercher à doter les noyaux de calcul de fonctionnalités permettant de détecter des erreurs silencieuses, avec des checksums ou avec des prédicteurs par exemple. Cette dernière fonctionnalité est cruciale car bien que moins probable, une erreur silencieuse peut corrompre le résultat de l'application et il ne sera pas possible de restaurer l'exécution avec les approches réalisées.

Enfin le redémarrage local a tendance à naturellement créer un déséquilibre de charge, et l'on pourra parfois perdre le bénéfice acquis en choisissant de garder en vie

les autres nœuds. Il est important dans ce cas de doter StarPU-MPI d'un mécanisme de rééquilibrage de charge, qui serait de toutes façons bénéfique même sans se préoccuper de la tolérance aux pannes, dans le cas d'applications présentant des déséquilibres dynamiques, telles que les codes AMR. Néanmoins la distribution statique des données et la soumission asynchrone, qui sont des propriétés intrinsèques à StarPU-MPI, réduisent grandement le champs d'action quand il s'agit de rééquilibrer la charge des nœuds, et donc le besoin de recherche pour concilier les deux aspects est indéniable. On ne pourrait néanmoins pas utiliser facilement le rééquilibrage de charge afin de pouvoir continuer l'exécution sur uniquement les $N - k$ nœuds survivants lorsque k nœuds tombent en panne, selon un redémarrage de type shrink. Le redémarrage global serait dans ce cas plus facile à implémenter, pour pouvoir forcer les nœuds survivants à exécuter des tâches qui ont déjà été soumises sur les nœuds perdus, de leur point de vue. Mais on pourrait aussi forcer un nœud sain à redémarrer, pour qu'il récupère la charge du nœud perdu en plus de sa propre charge, puis laisser un mécanisme de rééquilibrage de charge réagir.

-=-=-=-=-

Pour terminer, il est intéressant d'observer la cohérence des checkpoints qui est obtenue en les insérant dans une sémantique STF/SPMD. On avait pu voir que des checkpoints coordonnés imposent de bloquer l'exécution de l'ensemble des nœuds. On fait cela afin de capturer l'état global dans lequel est un système à un instant t, car un état par lequel est passé le programme est forcément un état cohérent. Les checkpoints non-coordonnés mettent en évidence qu'il existe une multitude d'états globaux cohérents, et proposent de sauver un état cohérent par lequel le système global n'est pas forcément passé. Mais les nœuds en eux-mêmes doivent tout de même être interrompus afin de réaliser une sauvegarde cohérente de l'application locale. Avec des checkpoints de niveau application, on interrompt également l'application, car l'on veut un état cohérent, et un état par lequel passe un programme est forcément cohérent. Lorsque l'on insère des checkpoints dans une application avec la sémantique STF/SPMD, on se rend compte qu'au sein même d'un programme local, il est possible de définir des états cohérents par lesquels l'application elle-même n'est jamais passée.

Bibliographie

[1] "ULFM Specification" (Feb 2017), http://fault-tolerance.org/ulfm/ulfm-specification/

[2] Adam, J., Besnard, J.B., Malony, A.D., Shende, S., Pérache, M., Carribault, P., Jaeger, J. : "Transparent High-Speed Network Checkpoint/Restart in MPI". In : Proceedings of the 25th European MPI Users' Group Meeting on - EuroMPI'18. pp. 1–11. ACM Press, Barcelona, Spain (2018). https://doi.org/10.1145/3236367.3236383, http://dl.acm.org/citation.cfm?doid=3236367.3236383

[3] Ahn, J. : "Communication-induced checkpointing with message logging beyond the piecewise deterministic (pwd) model for distributed systems". Electronics **10**(12) (2021). https://doi.org/10.3390/electronics10121428, https://www.mdpi.com/2079-9292/10/12/1428

[4] Alvisi, L., Marzullo, K. : "Message logging : pessimistic, optimistic, causal, and optimal". IEEE Transactions on Software Engineering **24**(2), 149–159 (Feb 1998). https://doi.org/10.1109/32.666828, conference Name : IEEE Transactions on Software Engineering

[5] Ashraf, R.A., Hukerikar, S., Engelmann, C. : "Shrink or Substitute : Handling Process Failures in HPC Systems Using In-Situ Recovery". In : 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 178–185 (Mar 2018). https://doi.org/10.1109/PDP2018.2018.00032

[6] Augonnet, C. : "StarPU : un support exécutif unifié pour les architectures multicoeurs hétérogènes". In : 19èmes Rencontres Francophones du Parallélisme (Sep 2009), https://hal.inria.fr/inria-00411581/document

[7] Augonnet, C., Aumage, O., Furmento, N., Thibault, S., Namyst, R. : "StarPU-MPI : Task Programming over Clusters of Machines Enhanced with Accelerators". report, INRIA (May 2014), https://hal.inria.fr/hal-00992208/document

[8] Aupy, G., Robert, Y., Vivien, F. : "Assuming Failure Independence : Are We Right to be Wrong ?". In : 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 709–716. IEEE, Honolulu, HI, USA (Sep

2017). https://doi.org/10.1109/CLUSTER.2017.24, http://ieeexplore.ieee.org/document/8049007/

[9] Barnafi, N.A., Huynh, N.M.M., Pavarino, L.F., Scacchi, S. : "Parallel nonlinear solvers in computational cardiac electrophysiology.". IFAC-PapersOnLine **55 (20)**, 187–192 (2022). https://doi.org/https://doi.org/10.1016/j.ifacol.2022.09.093, https://dx.doi.org/https://doi.org/10.1016/j.ifacol.2022.09.093

[10] Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S. : "FTI : High performance Fault Tolerance Interface for hybrid systems". In : SC '11 : Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (Nov 2011). https://doi.org/10.1145/2063384.2063427, iSSN : 2167-4337

[11] Benoit, A., Hérault, T., Fèvre, V.L., Robert, Y. : "Replication is more efficient than you think". In : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–14 (2019)

[12] Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J. : "Post-failure recovery of MPI communication capability : Design and rationale". The International Journal of High Performance Computing Applications **27**(3), 244–254 (Aug 2013). https://doi.org/10.1177/1094342013488238, https://doi.org/10.1177/1094342013488238

[13] Bland, W., Lu, H., Seo, S., Balaji, P. : "Lessons learned implementing user-level failure mitigation in mpich". In : 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 1123–1126 (2015). https://doi.org/10.1109/CCGrid.2015.51

[14] Bosilca, G. : "Spurious errors : lack of mpi progress and failure detection". fault-tolerance.org/2021/11/12/sc21-tutorial/

[15] Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Robert, Y., Sens, P., Dongarra, J. : "Failure Detection and Propagation in HPC Systems". In : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 27:1–27:11. SC '16, IEEE Press, Piscataway, NJ, USA (2016), http://dl.acm.org/citation.cfm?id=3014904.3014941, event-place : Salt Lake City, Utah

[16] Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P., Cappello, F. : "MPICH-V Project : A Multiprotocol Automatic Fault-Tolerant MPI". Int. J. High Perform. Comput. Appl. **20**(3), 319–333 (Aug 2006). https://doi.org/10.1177/1094342006067469, http://dx.doi.org/10.1177/1094342006067469

[17] Bouteiller, A. : “Spurious errors : lack of mpi progress and failure detection”. fault-tolerance.org/2020/01/21/spurious-errors-lack-of-mpi-progress-and-failure-detection/

[18] Bouteiller, A., Bosilca, G., Dongarra, J. : “Redesigning the message logging model for high performance”. Concurrency and Computation : Practice and Experience **22**(16), 2196–2211 (Nov 2010). https://doi.org/10.1002/cpe.1589, http://doi.wiley.com/10.1002/cpe.1589

[19] Buntinas, D., Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., Cappello, F. : “Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols”. Future Generation Computer Systems **24**(1), 73–84 (Jan 2008). https://doi.org/10.1016/j.future.2007.02.002, https://linkinghub.elsevier.com/retrieve/pii/S0167739X07000258

[20] Cao, C., Herault, T., Bosilca, G., Dongarra, J. : “Design for a Soft Error Resilient Dynamic Task-Based Runtime”. In : 2015 IEEE International Parallel and Distributed Processing Symposium. pp. 765–774. IEEE, Hyderabad, India (May 2015). https://doi.org/10.1109/IPDPS.2015.81, http://ieeexplore.ieee.org/document/7161563/

[21] Cappello, F., Guermouche, A., Snir, M. : “On Communication Determinism in Parallel HPC Applications”. In : 2010 Proceedings of 19th International Conference on Computer Communications and Networks. pp. 1–8 (Aug 2010). https://doi.org/10.1109/ICCCN.2010.5560143

[22] Cappello, F. : “Fault Tolerance in Petascale/ Exascale Systems : Current Knowledge, Challenges and Research Opportunities”. The International Journal of High Performance Computing Applications **23**(3), 212–226 (Aug 2009). https://doi.org/10.1177/1094342009106189, http://journals.sagepub.com/doi/10.1177/1094342009106189

[23] Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., Snir, M. : “Toward Exascale Resilience”. The International Journal of High Performance Computing Applications **23**(4), 374–388 (Nov 2009). https://doi.org/10.1177/1094342009347767, http://journals.sagepub.com/doi/10.1177/1094342009347767

[24] Chakraborty, S., Laguna, I., Emani, M., Mohror, K., Panda, D.K., Schulz, M., Subramoni, H. : “ER einit : Scalable and efficient fault-tolerance for bulk-synchronous MPI applications”. Concurrency and Computation : Practice and Experience **32**(3) (Feb 2020). https://doi.org/10.1002/cpe.4863, https://onlinelibrary.wiley.com/doi/10.1002/cpe.4863

[25] Chakraborty, S., Laguna, I., Emani, M., Mohror, K., Panda, D.K., Schulz,

M., Subramoni, H. : “Ereinit : Scalable and efficient fault-tolerance for bulk-synchronous mpi applications”. Concurrency and Computation : Practice and Experience **32**(3), e4863 (2020). https://doi.org/https://doi.org/10.1002/cpe.4863, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4863, e4863 cpe.4863

[26] Chandy, K.M., Lamport, L. : “Distributed snapshots : Determining global states of distributed systems”. ACM Trans. Comput. Syst. **3**(1), 63–75 (feb 1985). https://doi.org/10.1145/214451.214456, https://doi.org/10.1145/214451.214456

[27] Coti, C. : “Fault Tolerance Techniques for Distributed, Parallel Applications”. Innovative Research and Applications in Next-Generation High Performance Computing pp. 221–252 (2016). https://doi.org/10.4018/978-1-5225-0287-6.ch009, https://www.igi-global.com/chapter/fault-tolerance-techniques-for-distributed-parallel-applications/159047

[28] Czechowski, K., Vuduc, R. : “A Theoretical Framework for Algorithm-Architecture Co-design”. In : 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 791–802 (May 2013). https://doi.org/10.1109/IPDPS.2013.99, iSSN : 1530-2075

[29] Dauwe, D., Pasricha, S., Maciejewski, A.A., Siegel, H.J. : “An Analysis of Resilience Techniques for Exascale Computing Platforms”. In : 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 914–923 (May 2017). https://doi.org/10.1109/IPDPSW.2017.41

[30] Di, S., Robert, Y., Vivien, F., Cappello, F. : “Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model”. IEEE Transactions on Parallel and Distributed Systems **28**(1), 244–259 (Jan 2017). https://doi.org/10.1109/TPDS.2016.2546248, conference Name : IEEE Transactions on Parallel and Distributed Systems

[31] Dongarra, J., Herault, T., Robert, Y. : “Revisiting the Double Checkpointing Algorithm”. In : 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. pp. 706–715. IEEE, Cambridge, MA, USA (May 2013). https://doi.org/10.1109/IPDPSW.2013.11, http://ieeexplore.ieee.org/document/6650947/

[32] Du, P., Bouteiller, A., Bosilca, G., Herault, T., Dongarra, J. : “Algorithm-based Fault Tolerance for Dense Matrix Factorizations”. In : Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 225–234. PPoPP ’12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2145816.2145845, http://doi.acm.org/10.1145/2145816.2145845, event-place : New Orleans, Louisiana, USA

[33] Du, Y., Marchal, L., Pallez, G., Robert, Y. : "Optimal Checkpointing Strategies for Iterative Applications". IEEE Transactions on Parallel and Distributed Systems **33**(3), 507–522 (Mar 2022). https://doi.org/10.1109/TPDS.2021.3099440, https://hal.inria.fr/hal-03338278

[34] Egwutuoha, I.P., Levy, D., Selic, B., Chen, S. : "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems". The Journal of Supercomputing **65**(3), 1302–1326 (Sep 2013). https://doi.org/10.1007/s11227-013-0884-0, http://link.springer.com/10.1007/s11227-013-0884-0

[35] Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B. : "A Survey of Rollback-recovery Protocols in Message-passing Systems". ACM Comput. Surv. **34**(3), 375–408 (Sep 2002). https://doi.org/10.1145/568522.568525, http://doi.acm.org/10.1145/568522.568525

[36] Engelmann, C., Ong, H., Scott, S.L. : "The Case for Modular Redundancy in Large-Scale High Performance Computing Systems". In : Proceedings of the 8th IASTED international conference on parallel and distributed computing and networks (PDCN). p. 7 (2009)

[37] Fagg, G.E., Dongarra, J.J. : "FT-MPI : Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World". In : Dongarra, J., Kacsuk, P., Podhorszki, N. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. pp. 346–353. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2000)

[38] Fagg, G.E., Gabriel, E., Bosilca, G., Angskun, T., Chen, Z., Pjesivac-grbovic, J., London, K., Dongarra, J.J. : "Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems". In : In Proceeding of International Supercomputer Conference (ICS (2003)

[39] Ferreira, K., Stearley, J., Laros, III, J.H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D. : "Evaluating the Viability of Process Replication Reliability for Exascale Systems". In : Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 44:1–44:12. SC '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2063384.2063443, http://doi.acm.org/10.1145/2063384.2063443, event-place : Seattle, Washington

[40] Georgakoudis, G., Guo, L., Laguna, I. : "Reinit++ : Evaluating the Performance of Global-Restart Recovery Methods For MPI Fault Tolerance". arXiv:2102.06896 [cs] (Feb 2021), http://arxiv.org/abs/2102.06896, arXiv : 2102.06896

[41] Graham, R.L., Barrett, B.W., Shipman, G.M., Woodall, T.S., Bosilca, G. : "Open MPI : A High Performance, Flexible Implementation of MPI Point-to-Point Communications". Parallel Processing Letters **17**(01), 79–88 (Mar 2007).

https://doi.org/10.1142/S0129626407002880, http://www.worldscientific.com/doi/abs/10.1142/S0129626407002880

[42] Gropp, W., Lusk, E. : "Fault Tolerance in Message Passing Interface Programs". The International Journal of High Performance Computing Applications **18**(3), 363–372 (Aug 2004). https://doi.org/10.1177/1094342004046045, http://journals.sagepub.com/doi/10.1177/1094342004046045

[43] Guermouche, A., Ropars, T., Brunet, E., Snir, M., Cappello, F. : "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications". In : 2011 IEEE International Parallel Distributed Processing Symposium. pp. 989–1000 (May 2011). https://doi.org/10.1109/IPDPS.2011.95

[44] Guo, L., Georgakoudis, G., Parasyris, K., Laguna, I., Li, D. : "MATCH : An MPI Fault Tolerance Benchmark Suite" (Feb 2021), http://arxiv.org/abs/2102.06894, arXiv:2102.06894 [cs]

[45] Gupta, S., Patel, T., Engelmann, C., Tiwari, D. : "Failures in large scale systems : long-term measurement, analysis, and implications". In : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '17. pp. 1–12. ACM Press, Denver, Colorado (2017). https://doi.org/10.1145/3126908.3126937, http://dl.acm.org/citation.cfm?doid=3126908.3126937

[46] Hukerikar, S., Engelmann, C. : "Resilience Design Patterns : A Structured Approach to Resilience at Extreme Scale". Supercomputing Frontiers and Innovations **4**(3) (Sep 2017). https://doi.org/10.14529/jsfi170301, http://arxiv.org/abs/1708.07422, arXiv : 1708.07422

[47] Karablieh, F., Bazzi, R., Hicks, M. : "Compiler-assisted heterogeneous checkpointing". In : Proceedings 20th IEEE Symposium on Reliable Distributed Systems. pp. 56–65 (2001). https://doi.org/10.1109/RELDIS.2001.969743

[48] Keller, K., Bautista-Gomez, L. : "Application-Level Differential Checkpointing for HPC Applications with Dynamic Datasets". In : 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 52–61 (May 2019). https://doi.org/10.1109/CCGRID.2019.00015

[49] Liu, R.T., Chen, Z.N. : "A Large-Scale Study of Failures on Petascale Supercomputers". Journal of Computer Science and Technology **33**(1), 24–41 (Jan 2018). https://doi.org/10.1007/s11390-018-1806-7, http://link.springer.com/10.1007/s11390-018-1806-7

[50] Losada, N., Bosilca, G., Bouteiller, A., González, P., Martín, M.J. : "Local rollback for resilient MPI applications with application-level checkpointing and message logging". Future Generation Computer Systems **91**, 450–464 (Feb

2019). https://doi.org/10.1016/j.future.2018.09.041, http://www.sciencedirect.com/science/article/pii/S0167739X18303443

[51] Moore, G.E., et al. : "Cramming more components onto integrated circuits" (1965)

[52] Naksinehaboon, N., Liu, Y., Leangsuksun, C., Nassar, R., Paun, M., Scott, S.L. : "Reliability-aware approach : An incremental checkpoint/restart model in hpc environments". In : 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID). pp. 783–788 (2008). https://doi.org/10.1109/CCGRID.2008.109

[53] Netzer, R.H.B., Jian Xu : "Necessary and sufficient conditions for consistent global snapshots". IEEE Transactions on Parallel and Distributed Systems **6**(2), 165–169 (Feb 1995). https://doi.org/10.1109/71.342127, conference Name : IEEE Transactions on Parallel and Distributed Systems

[54] Nicolae, B., Moody, A., Gonsiorowski, E., Mohror, K., Cappello, F. : "VeloC : Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale". In : 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 911–920 (May 2019). https://doi.org/10.1109/IPDPS.2019.00099, iSSN : 1530-2075

[55] Papadopoulos, L., Soudris, D., Kessler, C., Ernstsson, A., Ahlqvist, J., Vasilas, N., Papadopoulos, A.I., Seferlis, P., Prouveur, C., Haefele, M., Thibault, S., Salamanis, A., Ioakimidis, T., Kehagias, D. : "EXA2PRO : A Framework for High Development Productivity on Heterogeneous Computing Systems". IEEE Transactions on Parallel and Distributed Systems (Aug 2021). https://doi.org/10.1109/TPDS.2021.3104257, https://hal.inria.fr/hal-03318644

[56] Posner, J. : Load Balancing, Fault Tolerance, and Resource Elasticity for Asynchronous Many-Task Systems. Ph.D. thesis, Kassel, Universität Kassel, Fachbereich Elektrotechnik / Informatiik (12 2021)

[57] Schneider, D. : "The exascale era is upon us : The frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second". IEEE Spectrum **59**(1), 34–35 (2022). https://doi.org/10.1109/MSPEC.2022.9676353

[58] Schroeder, B., Gibson, G. : "A large-scale study of failures in high-performance computing systems". In : International Conference on Dependable Systems and Networks (DSN'06). pp. 249–258 (Jun 2006). https://doi.org/10.1109/DSN.2006.5, iSSN : 2158-3927

[59] Strom, R., Yemini, S. : "Optimistic recovery in distributed systems". ACM Transactions on Computer Systems (TOCS) **3**(3), 204–226 (Aug 1985). https://doi.org/10.1145/3959.3962, http://dl.acm.org/doi/10.1145/3959.3962

[60] Subasi, O., Martsinkevich, T.V., Zyulkyarov, F., Unsal, O.S., Labarta, J., Cappello, F. : "Unified fault-tolerance framework for hybrid task-parallel message-passing applications". The International Journal of High Performance Computing Applications **32**, 641 – 657 (2018)

[61] Susan Blackford : "The Two-dimensional Block-Cyclic Distribution" (May 1997), http://www.netlib.org/scalapack/slug/node75.html

[62] Tang, X., Zhai, J., Yu, B., Chen, W., Zheng, W. : "Self-Checkpoint : An In-Memory Checkpoint Method Using Less Space and Its Practice on Fault-Tolerant HPL". In : Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 401–413. ACM, Austin Texas USA (Jan 2017). https://doi.org/10.1145/3018743.3018745, https://dl.acm.org/doi/10.1145/3018743.3018745

[63] Tessier, F., Vishwanath, V., Jeannot, E. : "TAPIOCA : An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers". In : 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 70–80. IEEE, Honolulu, HI, USA (Sep 2017). https://doi.org/10.1109/CLUSTER.2017.80, http://ieeexplore.ieee.org/document/8048918/

[64] Vasavada, M., Mueller, F., Hargrove, P.H., Roman, E. : "Comparing different approaches for Incremental Checkpointing : The Showdown". In : Linux Symposium. p. 11 (2011)

[65] Walker, D.W., Dongarra, J.J. : "MPI : a standard message passing interface". Supercomputer **12**, 56–68 (1996)

[66] Woo, N., Jung, H., Yeom, H.Y., Park, T., Park, H. : "MPICH-GF : Transparent Checkpointing and Rollback-Recovery for Grid-Enabled MPI Processes". IEICE Transactions **87-D**, 1820–1828 (2004)

[67] Young, J.W. : "A First Order Approximation to the Optimum Checkpoint Interval". Commun. ACM **17**(9), 530–531 (Sep 1974). https://doi.org/10.1145/361147.361115, http://doi.acm.org/10.1145/361147.361115

[68] Zheng, G., Shi, L., Kale, L. : "FTC-Charm++ : an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI". In : 2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935). pp. 93–103 (Sep 2004). https://doi.org/10.1109/CLUSTR.2004.1392606, iSSN : 1552-5244

[69] Zheng, G., Xiang Ni, Kale, L.V. : "A scalable double in-memory checkpoint and restart scheme towards exascale". In : IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012). pp. 1–6. IEEE, Boston, MA, USA (Jun 2012). https://doi.org/10.1109/DSNW.2012.6264677, http://ieeexplore.ieee.org/document/6264677/

Réplication de donnée pour la tolérance aux pannes dans un support d'exécution distribué à base de tâches

Résumé : À mesure que la puissance de calcul des nouveaux supercalculateurs augmente, leur fiabilité décroît inexorablement. En effet les limites sont repoussées en augmentant le nombre de composants ainsi que leur complexité, et les systèmes de calcul expérimentent des défaillances au quotidien. La problématique est donc de pouvoir se prémunir des pannes, tout en limitant l'impact sur les performances qu'impose un mécanisme de tolérance aux pannes. Par ailleurs, la complexité de l'architecture des supercalculateurs rend plus difficile leur programmation, ce à quoi tentent de répondre les supports d'exécution à base de tâches comme StarPU. Les travaux présentés ici proposent une solution de tolérance aux pannes adaptée à StarPU. Le modèle de programmation STF utilisé dans StarPU nous permet de créer des sauvegardes asynchrones qui n'interrompent pas le calcul, et qui ne nécessitent aucune synchronisation entre les nœuds de calcul ; ce comportement est atteint en insérant statiquement des requêtes de sauvegarde dans le code d'application, correspondant à une solution de checkpoint de niveau application. Également, gérer des sauvegardes à travers StarPU permet de mettre en commun les données de calcul et les données de sauvegarde, ce qui nous permet de réduire considérablement la quantité de données qui doivent effectivement être sauvegardées. Nous avons exploité cet effet en utilisant les autres nœuds de calcul comme support de sauvegarde, ainsi qu'un mécanisme de message logging afin de redémarrer uniquement le nœud en panne. L'efficacité de cette solution a été évaluée avec une application de décomposition de Cholesky, et nous avons pu voir dans un cas idéal que notre approche permet de tolérer les pannes considérées sans qu'aucune sauvegarde ne soit effectuée en pratique.
Mots-clés : Tolérance aux pannes, Checkpoint, Support d'exécution à base de tâches

---

Data replication for failure tolerance in a task-based runtime system

Abstract: While computing power of systems grows, their reliabity decreases inevitably. Indeed, performance is achieved by leveraging components quantity and complexity, therefore computing systems are subject to failures on a daily basis. The problem is to use a mechanism to tolerate failures while having the least impact on performance. Moreover, supercomputers archichecture become more and more complex, and so becomes their coding. Data-based runtime systems such as StarPU are responding to this problematic. This thesis proposes a dedicated failure tolerance protocol to StarPU. The STF programming model used in StarPU allows to create consistent coordinated non-blocking asynchronous checkpoints very simply, by inserting checkpoint requests statically in the source code, like an application-based checkpoint solution. Furthermore, managing the checkpoints inside StarPU allows to use the synergy between computing data and checkpoint data, allowing to significantly reduce the amount of data that needs to be saved. We exploit this effect by choosing to save checkpoints on the other computing nodes, and by performing local rollback using message logging. The efficiency of our proposal is evaluated with a Cholesky decomposition application. We also show that with a particular setting for this application, our approach allows to tolerate the failure corresponding to our hypothesis without having any data actually replicated on other nodes. This is done by using the fact that the application already replicates enough data due to the computation needs, while with our approach we are able to exploit these data as checkpoint data.
Keywords: Failure tolerance, Checkpoints, Task-based runtime system

---

Unité de recherche :
Laboratoire Bordelais de Recherche en Informatique (LaBRI) UMR 5800, Université de Bordeaux, 33400 Talence, France.