Tolérance aux pannes dans l'exécution distribuée de graphes de tâches

Romain Lion

▶ **To cite this version:**

Romain Lion. Tolérance aux pannes dans l'exécution distribuée de graphes de tâches. COMPAS 2019 - Conférence d'informatique en Parallélisme, Architecture et Système, Jun 2019, Anglet, France. hal-02296118

HAL Id: hal-02296118
https://inria.hal.science/hal-02296118v1

Submitted on 24 Sep 2019

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

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

Compas'2019 : Parallélisme / Architecture / Système
LIUPPA - IUT de Bayonne, France, du 24 au 28 juin 2019

Tolérance aux pannes dans l'exécution distribuée de graphes de tâches

Romain Lion – romain.lion@inria.fr

Équipe STORM – INRIA Bordeaux Sud-Ouest
200 Avenue de la Vieille Tour – 33405 Talence – France

Résumé
Les plus grands supercalculateurs rassemblent un nombre toujours croissant d'unités de calcul, ce qui augmente d'autant le taux de pannes. Des méthodes de checkpoint/restart ont été proposées pour éviter que, lorsqu'un nœud est totalement perdu, l'on doive reprendre l'exécution de l'application depuis son départ. Ces méthodes sont cependant en général transparentes et ne profitent pas d'informations connues sur le comportement de l'application. Inversement, le paradigme de programmation par graphe de tâches fournit l'opportunité de proposer des méthodes de checkpoint/restart bien plus judicieuses. Nous proposons ainsi une approche qui permettra de ne sauvegarder que les données utiles en cohérence avec les communications de l'application, de supporter un redémarrage local, tout en exhibant une interface de programmation simple intégrée à la programmation de graphe de tâches.

Mots-clés : Tolérance aux pannes, Graphe de tâches, MPI, Support d'exécution

---

1. Introduction

Si nos machines devraient atteindre l'exascale d'ici 2022, nos outils ne sont en revanche pas totalement prêts à accueillir cette avancée technologique. En effet la puissance des fermes de calcul ne pourra être démultipliée qu'en augmentant davantage le nombre d'unités de calcul.
Une des problématiques soulevée par ce grand nombre d'unités est que le temps moyen entre deux pannes ne devient plus négligeable. Le système sera alors susceptible de perdre une unité de calcul plus fréquemment, plusieurs pouvant intervenir au cours d'un travail de quelques heures ; il faut par conséquent rendre nos outils résilients.
Les solutions logicielles visant l'exascale se doivent donc d'inclure un système de tolérance aux pannes. Dans le cas de StarPU, qui est un outil pouvant distribuer l'exécution d'un graphe de tâches avec MPI, il est possible d'exprimer de puissantes hypothèses pour l'implémentation de la résilience, ce que ne peuvent émettre les protocoles génériques et universels. Ceci nous amène à dire qu'une implémentation de tolérance aux pannes adaptée à StarPU est utile pour entraîner moins de surcharge, permettant ainsi de passer à l'échelle.
Nous présentons d'abord la solution de résilience dont nous nous sommes inspirés, avant de voir les implémentations MPI adaptées à notre besoin pour ensuite présenter l'outil StarPU. En second nous détaillons nos choix pour le développement d'un modèle de programmation bénéficiant des avantages qu'offre StarPU et répondant à la problématique, avant d'illustrer l'implémentation à venir. Enfin nous évoquons un ensemble d'optimisations qui seront réalisables après implémentation.

2. Contexte et état de l'Art

2.1. Contexte

On peut tout d'abord distinguer les *pannes* et les *fautes*. Une *faute* est par exemple une erreur de calcul, ou tout phénomène retournant un résultat erroné demandant une analyse pour être détecté. Une *panne* est un phénomène bloquant qui porte sur un élément physique remettant en cause son fonctionnement, et ne retourne par conséquent aucun résultat. Nous souhaitons traiter ici uniquement les pannes de nœuds entiers, qui sont des phénomènes pouvant être détectés par les couches de communication et qui répondent à la problématique soulevée par les réseaux exascales.

Nous nous intéressons à la résilience par substitution. Après une panne, il faut recréer un processus substitut qui soit dans un état proche du processus perdu au moment de sa perte. Cette proximité est temporelle : après un certain temps le processus substitut doit retrouver l'état du processus perdu. Il peut alors être nécessaire de reproduire les évènements extérieurs au processus : dans le cas d'un système communiquant par messages, il faut rejouer les messages impactant ce processus ce qui est suffisant pour obtenir un modèle respectant le déterminisme de l'application. En d'autres termes, il est possible de représenter un processus par une machine à état qui subit une série d'événements, ici des communications de messages[6].

2.2. Checkpoint/Restart et Checkpoint/Rollback partiel

Pour restaurer l'état d'un processus, il est nécessaire d'avoir accès à certaines données. Le checkpoint peut alors être vu comme un paquet contenant ces données. Sans hypothèse de contexte, il faut sauvegarder l'ensemble de la mémoire et la position dans le programme, mais cette pratique est coûteuse. En pratique, seul un sous-ensemble des données en mémoire est réellement nécessaire pour restaurer la cohérence et le déterminisme de l'application, réduisant considérablement la taille des checkpoints. Dans certains cas, notamment si la prise des checkpoint n'est pas coordonnée lors d'instants logiques cohérents entre les nœuds, il peut être nécessaire de garder certains messages pour les rejouer en cas de panne [5]. Il faut donc sauvegarder de manière pérenne les checkpoints et les messages, et les solutions utilisent généralement un stockage sur disque à cet effet.

Une fois ces données sauvegardées, il faut pouvoir les réutiliser selon un modèle de reprise d'exécution en cas de panne. Le modèle *Restart* implique de redémarrer l'ensemble des processus vers un état global cohérent. En utilisant cette méthode avec des checkpoints non-coordonnés, il est nécessaire de garder un journal de message si l'on ne veut pas subir un effet domino pour retrouver un état global cohérent : ce phénomène peut arriver lorsque le seul état cohérent atteignable est l'état initial. En opposition au Restart, on parle de *Rollback partiel* lorsque l'on ne rembobine que le processus perdu. Cette méthode a l'avantage d'être moins énergivore et potentiellement plus adaptable à l'application que la précédente, mais est néanmoins plus intrusive dans le code de l'application.

2.3. Solutions de tolérance aux pannes pour MPI

Bien que le standard MPI ne définit pas encore de protocole ou d'outils pour rendre les applications tolérantes aux pannes, de nombreuses implémentations proposent des solutions à deux niveaux différents.

Certaines implémentations MPI [4] [10] proposent de fournir un support de tolérance aux pannes de *niveau système* et donc transparent à l'utilisateur. Néanmoins ces implémentations génériques ne peuvent pas tirer parti de l'ensemble des avantages fournis par certaines applications où des hypothèses fortes peuvent être émises, limitant les résultats attendus.

D'autres implémentations MPI [7] [3] proposent au contraire des outils de *niveau utilisateur*, permettant à ce dernier d'implémenter un protocole de résilience adapté à ses besoins. Le code devient alors dépendant de l'implémentation MPI. L'implémentation ULFM (User Level Fault Mitigation) [3] possède une spécification en cours de standardisation et prenant en compte les efforts passés. Celle-ci propose à l'utilisateur des outils pour détecter la perte d'un nœud MPI, pour mettre en place des procédures de remédiation et pour diffuser l'information de panne à l'aide de codes d'erreur spécifiques. Si la standardisation d'ULFM n'est pas actée, il s'agit cependant du choix le plus pertinent à l'heure actuelle.

2.4. StarPU, un modèle d'exécution basé sur des tâches

StarPU est une bibliothèque développée depuis 2008 au sein de notre équipe, et qui propose un moteur d'exécution à base de tâches. L'intention initiale est de proposer des tâches *variantes* : chaque tâche peut posséder plusieurs implémentations, afin d'être potentiellement exécutée sur plusieurs types d'unité de calcul d'une machine hétérogène. Un ordonnanceur est en charge d'assigner une tâche à une unité de calcul plutôt qu'à une autre, selon une politique paramétrable [8]. StarPU permet également de distribuer le calcul sur un réseau de machines en utilisant MPI selon un modèle maître-esclave, ou bien dans un mode totalement réparti [2] selon un modèle SPMD. Nous nous intéressons dans cet article à ce mode réparti décentralisé, dans lequel la distribution des tâches sur les nœuds est réalisée grâce à l'attribution d'un nœud propriétaire pour chaque donnée. Par défaut un nœud exécutera uniquement les tâches qui modifient les données dont il est propriétaire, mais il est possible de définir une autre politique, pourvu qu'elle soit déterministe.
On a donc une application qui suit un modèle de programmation SPMD (Single Program - Multiple Data) avec des données spécifiques à chaque nœud MPI, alors que la soumission du même graphe de tâches est effectuée séquentiellement sur chaque machine (Sequential Task Flow), chaque machine décidant de manière déterministe quelle portion du graphe elle va exécuter. Le programme de l'application étant séquentiel, la soumission du graphe de tâches est également séquentielle, ce qui nous apporte des propriétés intéressantes qui sont exploitées par le modèle de programmation que nous allons présenter.

3. Solution de résilience pour StarPU

3.1. Présentation du protocole

Nous avons choisi de définir un protocole de résilience impliquant des checkpoints car il s'agit d'un paradigme permettant de bénéficier des propriétés héritées de StarPU. Cet outil étant basé sur des graphes de tâches, il nous offre des possibilités innovantes face aux techniques classiques ce qui peut limiter la surcharge provoquée par la résilience.
Pour la reprise de service après une panne, nous choisissons d'employer une technique de rollback partiel, qui peut être mise en place avec les outils fournis par l'implémentation ULFM. Ceci est rendu possible en grande partie grâce au paradigme de graphe de tâches proposé par StarPU. Nous choisissons dans un premier temps de stocker les checkpoints non pas sur disque, mais sur un nœud backup. En effet, chaque nœud MPI se voit attribuer un nœud backup vers lequel il communique son checkpoint, tout en étant lui même le backup d'un autre nœud. Le checkpoint est donc stocké à un seul endroit, amenant une limite dont il faut avoir conscience ; si un nœud meurt, et si son backup tombe en panne avant de pouvoir utiliser le checkpoint, on obtiendra un blocage du système. Néanmoins nous choisissons dans un premier temps de ne pas le considérer.
Enfin nous choisissons de travailler avec un principe de substitution de nœud. Ce dernier per-

met de restaurer le nœud MPI perdu sur une nouvelle machine en créant un nœud *substitut*, afin de garantir un nombre de travailleurs constant et d'éviter une redistribution des données.

3.2. Interface de programmation

L'utilisateur doit identifier les données nécessaires à la reconstitution de l'état du processus. Ces données peuvent être classées en deux catégories : les données enregistrées auprès de StarPU comme les matrices de calculs intermédiaires par exemple, et les données de contrôle de l'application, telle qu'une variable d'itération qui permet de situer la position dans l'exécution. La syntaxe d'une méthode checkpoint peut donc se résumer au passage d'un nombre variable de paramètres. Par exemple pour un cas simple où l'application se résume à une simple boucle 'for', et que le checkpoint est appelé en début de cette boucle, il suffit de sauvegarder la donnée de calcul et la variable d'itération pour pouvoir recréer plus tard l'état, comme montré dans l'échantillon de code ci-dessous.

```
starpu_ft_checkpoint(STARPU_DATA, &my_matrix, STARPU_VALUE, &i,
    sizeof(i)); // my_matrix est un resultat intermediaire a sauvegarder
    et i est la variable d'iteration.
```

Il est utile de noter que la soumission des tâches par StarPU étant séquentielle au niveau de chaque nœud, nous pouvons garantir que la création d'un checkpoint se fait selon les dépendances déduites du graphe de tâches. En effet une barrière locale doit être effectuée, afin de s'assurer de l'état des données lors du checkpoint : le checkpoint est sauvé après la fin de chaque tâche précédent l'appel, et aucune tâche n'est exécutée si elle est soumise après la soumission du checkpoint.
En plus de cela, l'utilisateur doit pouvoir charger un checkpoint lors de l'initialisation. Une syntaxe comme celle présentée ci-dessous permet de répondre à ce besoin.

```
starpu_ft_reload(&my_matrix, &i); // my_matrix et i doivent etre
    fournis dans le meme ordre que lors de l'appel de la fonction
    starpu_ft_checkpoint.
```

L'appel à ces fonctions doit être positionné afin conserver le déterminisme de l'application. Un exemple de code est disponible en annexe A.
L'implication de l'utilisateur peut donc se réduire à identifier les données significatives ainsi qu'à veiller à bien positionner la création de checkpoint et l'initialisation après une panne. Le reste du fonctionnement est déductible de la soumission séquentielle du graphe de tâches, et peut être donc fait de manière transparente vis-à-vis de l'utilisateur.

3.3. Coeur du fonctionnement

En plus de la prise de checkpoints, il est nécessaire de sauvegarder les messages pour pouvoir les rejouer lors de la refonte de l'état d'un processus. Nous proposons que chaque nœud sauvegarde automatiquement ses messages sortants ainsi qu'une trace des messages entrants, afin de pouvoir rejouer les émissions et ignorer les réceptions lorsque cela sera nécessaire.
L'appel à la fonction checkpoint est fait séquentiellement avec les soumissions de tâches, comme évoqué précédemment. De plus comme l'on a une sémantique SPMD, la prise des checkpoints est logiquement coordonnée entre les différents nœuds. En effet une propriété importante est que chaque nœud sait quand a lieu le checkpoint dans le code. En numérotant en interne chaque checkpoint, on peut simplifier la déduction de la séquence de messages à rejouer.
Après s'être assuré de l'initialisation du substitut, le nœud backup va déduire du graphe de

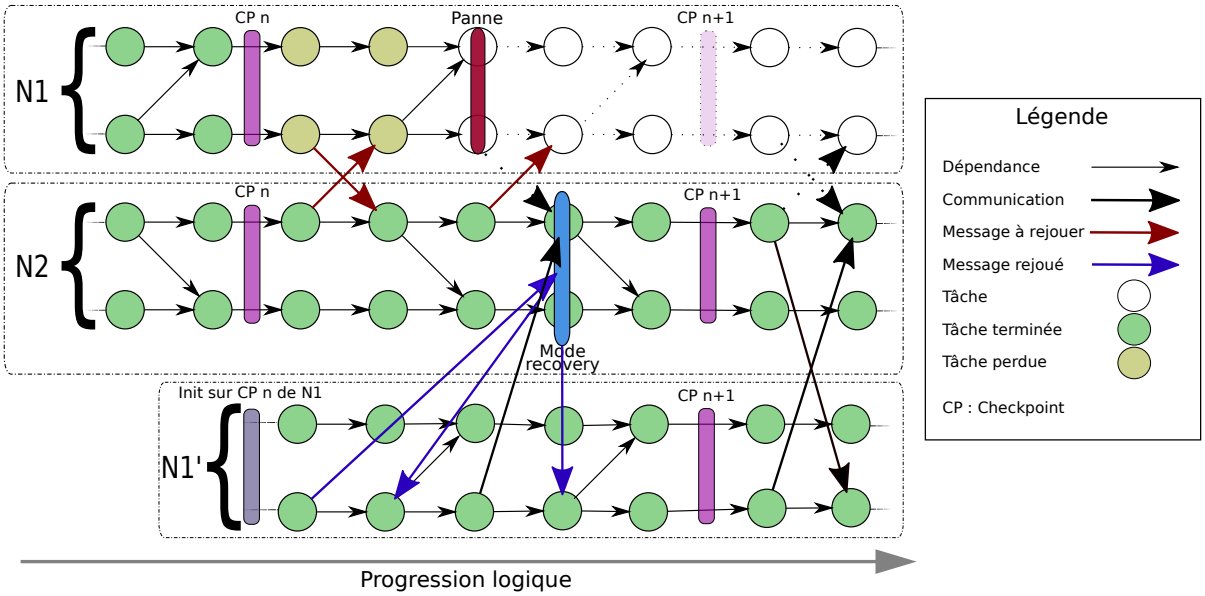

FIGURE 1 – Exemple de la procédure de résilience.

tâches les nœuds communicants potentiellement avec le substitut, leur envoyer le numéro de checkpoint, permettant à chacun de déduire l'état du substitut et ainsi rejouer des messages. Néanmoins cette étape de diffusion peut être coûteuse si les nœuds potentiellement impliqués sont nombreux ; le nombre de messages de diffusion à transmettre s'exprime par le nombre de correspondants qu'a un nœud entre deux checkpoints.

3.4. Déroulement d'une reprise après panne

Maintenant que les différents principes ont été présentés, nous proposons de les illustrer en décrivant les étapes au travers d'un schéma (figure 1).

1. Un checkpoint n a été fait sur le nœud N1. Ce dernier est stocké sur son backup (N2).
2. Une panne a lieu sur le nœud N1 et une signalisation est déclenchée.
3. Le réseau MPI est assaini en créant un substitut N1′. Lors de ce rétablissement, le nœud N2 entre en mode *recovery* et envoie les données du checkpoint n au substitut N1′ : Le nœud N1′ est désormais dans un état cohérent, initialisé sur le checkpoint n.
4. Néanmoins une séquence de message doit être rejouée. Pour cela le backup N2 contacte les nœuds qui doivent potentiellement renvoyer un message. Dans un souci de simplicité, la figure 1 ne représente que deux nœuds : le nœud N2 est le seul nœud devant potentiellement renvoyer des messages.
5. Les nœuds devant interagir avec le substitut N1′ entrent en mode *recovery*, afin de rejouer les messages. Sur la figure 1, il n'y a que N2 qui doit rejouer des messages.
6. Le nœud N1′ retrouve ainsi un état globalement cohérent et l'exécution peut continuer.

4. Travaux futurs

Le principe présenté demande à être évalué, mais sa simplicité laisse des marges de manœuvre pour des optimisations. Même si l'on ne provoque pas de barrières globales, et que la majeure

partie des opérations est réalisée par les nœuds de manière locale, il se pourrait que cet algorithme ne passe pas à l'échelle en l'état. C'est pourquoi nous proposons des améliorations tirant parti des avantages fournis par la sémantique des graphes de tâches que propose StarPU, afin d'aller au delà d'un simple principe de checkpoint.
Premièrement dans les travaux présentés, on garde une trace locale de l'ensemble des messages concernant le nœud. Il est clair que ceci ne peut passer à l'échelle si un grand nombre de messages entrent en jeu ; il est alors intéressant et réalisable d'implémenter un système de *garbage collector* permettant de supprimer les messages périmés.
Aussi les données de calcul contenues dans les checkpoints sont des données qui sont souvent partagées entre les nœuds pour des besoins algorithmiques. Il serait alors intéressant de prendre en compte ce paramètre lors de la prise de checkpoints. Ceci peut être déduit du graphe de tâches et permettrait de réduire la taille des checkpoints et la bande passante consommée lors de leur partage. De la même manière, si des données ne sont pas modifiées entre deux checkpoints, il peut être inutile de les sauver lors d'un nouveau checkpoint.
Un autre moyen d'améliorer le protocole serait de travailler non pas avec un backup mais plusieurs. En effet on a vu que travailler avec un seul backup implique d'être sensible à certains scénarios de panne. On a donc un intérêt à approfondir le sujet de la multiple réplication, ou bien stocker les checkpoints sur un disque en plus du backup à l'aide d'une comme librairie optimisée comme TAPIOCA[9].

5. Conclusion

Les ambitions du domaine HPC ne nous permettent plus de négliger les pannes des unités de calcul, et les outils doivent donc évoluer pour faire suivre les performances. Nous avons présenté dans cet article une solution de résilience basée sur des checkpoints, qui permet de bénéficier des avantages procurés par StarPU. Grâce au paradigme de soumission séquentielle du graphe de tâches, nous pouvons nous prémunir contre nombre d'obstacles identifiés lors de travaux sur la tolérance aux pannes, car cela nous apporte une connaissance locale à propos de ce qui est effectué sur les autres nœuds, sans pour autant saturer le réseau avec des communications inutiles à l'application. Il s'agit sans doute là du point le plus important apporté par le paradigme utilisé dans StarPU, et une approche simple telle que celle présentée dans cet article nous permet d'espérer de bons résultats.
Dans les travaux présentés, on ne se penche que sur un système de substitution lors de l'assainissement de réseau de machines. Cela implique d'avoir à disposition des machines *spares* prêtes à remplacer les machines en panne, afin de garantir un nombre de travailleurs constant. Néanmoins il est possible de travailler avec le sous-ensemble restant de machines, en réduisant le nombre total de travailleurs. Cette technique offre d'ailleurs des résultats intéressants face à la substitution [1]. StarPU fonctionne cependant pour l'instant selon un principe fortement dépendant du nombre de machines, et un problème se pose alors au niveau de la répartition de la charge. Il faudrait donc introduire un rééquilibrage de charge dynamique au sein de StarPU. Ce sujet est d'autant plus intéressant qu'un tel rééquilibrage peut être aussi utile dans un contexte sans panne.
Il est également envisageable de réduire l'implication de l'utilisateur. En effet il lui incombe de placer les appels aux checkpoints, ce qui peut être une tâche dont l'optimisation n'est pas triviale, car une répartition logiquement uniforme n'est pas nécessairement une répartition temporellement uniforme. Il peut être alors intéressant d'utiliser un compilateur *source-to-source* afin de définir une stratégie de placement des checkpoints automatisée, afin d'assurer une granularité uniforme et paramétrable par l'utilisateur.

Bibliographie

1. Ashraf (R. A.), Hukerikar (S.) et 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, mars 2018.
2. Augonnet (C.), Aumage (O.), Furmento (N.), Thibault (S.) et Namyst (R.). – *StarPU-MPI : Task Programming over Clusters of Machines Enhanced with Accelerators*. – report, INRIA, mai 2014.
3. Bland (W.), Bouteiller (A.), Herault (T.), Bosilca (G.) et Dongarra (J.). – Post-failure recovery of MPI communication capability : Design and rationale. *The International Journal of High Performance Computing Applications*, vol. 27, n3, août 2013, pp. 244–254.
4. Bouteiller (A.), Herault (T.), Krawezik (G.), Lemarinier (P.) et Cappello (F.). – MPICH-V Project : A Multiprotocol Automatic Fault-Tolerant MPI. *Int. J. High Perform. Comput. Appl.*, vol. 20, n3, août 2006, pp. 319–333.
5. Coti (C.). – Fault Tolerance Techniques for Distributed, Parallel Applications. *Innovative Research and Applications in Next-Generation High Performance Computing*, 2016, pp. 221–252.
6. Elnozahy (E. N. M.), Alvisi (L.), Wang (Y.-M.) et Johnson (D. B.). – A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, vol. 34, n3, septembre 2002, pp. 375–408.
7. Fagg (G. E.) et Dongarra (J. J.). – FT-MPI : Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. – In Dongarra (J.), Kacsuk (P.) et Podhorszki (N.) (édité par), *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, pp. 346–353. Springer Berlin Heidelberg, 2000.
8. Sergent (M.) et Archipoff (S.). – Modulariser les ordonnanceurs de tâches : une approche structurelle. – avril 2014.
9. Tessier (F.), Vishwanath (V.) et Jeannot (E.). – TAPIOCA : An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers. – In *CLUSTER 2017 - IEEE International Conference on Cluster Computing*, pp. 1–11, Honolulu, United States, septembre 2017. IEEE.
10. Woo (N.), Jung (H.), Yeom (H. Y.), Park (T.) et Park (H.). – MPICH-GF : Transparent Checkpointing and Rollback-Recovery for Grid-Enabled MPI Processes. *IEICE Transactions*, vol. 87-D, 2004, pp. 1820–1828.

Annexes

A. Exemple de code utilisant les fonctions pour la résilience

```
int main(int argc, char *argv[]) {
   // Declaration
   int i, istart;
   starpu_data_handle_t my_matrix;

   // Initialisation
   starpu_init(NULL);
   if (starpu_ft_has_rollback())
         starpu_ft_reload(&my_matrix, &istart);
   else {
         starpu_matrix_data_register(&my_matrix, ...);
         istart = 0;
   }

   // Soumission
  for (i = istart; i < N; i++) {
         starpu_ft_checkpoint(STARPU_DATA, &my_matrix, STARPU_VALUE, &i,
            sizeof(i));
         ...
         starpu_task_submit(..., my_matrix, ...);
         ...
   }

   // Terminaison
   starpu_shutdown();
}
```

Une des contribution des fonctions de résiliences apparaît dans la section d'initialisation. En effet nous utilisons la condition `starpu_ft_has_rollback` distinguant deux cas : soit l'on démarre l'application du début (test faux), soit l'on redémarre après une panne, lors d'un rollback (test vrai). Dans ce dernier cas, nous utilisons la fonction `starpu_ft_reload` afin d'initialiser certaines données à l'aide d'un checkpoint.
Les checkpoints sont créés lors de la phase de soumission, en début de boucle. Sa position permet de conserver le déterminisme de l'application lors de la restauration après une panne. Il aurait pu être placé en fin de boucle, en ayant à l'esprit qu'une modification doit être effectuée sur la variable d'itération afin de rester cohérent. Placer le checkpoint ailleurs dans la boucle peut être difficile voire impossible étant donné que la cohérence doit être conservée.