Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité

Samir Jafar

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : "Informatique : Systèmes et Logiciels"

préparée au laboratoire INFORMATIQUE ET DISTRIBUTION
dans le cadre de ***l'École Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique"***

présentée et soutenue publiquement

par

Samir Jafar

le 30 Juin 2006

Programmation des systèmes parallèles distribués : tolérance aux pannes, résilience et adaptabilité

Directeur de thèse : Denis TRYSTRAM

JURY

MR VAN DAT CUNG	INPG – France	Président
MR MOHAMED JEMNI	U. de Tunis – Tunisie	Rapporteur
MR PIERRE SENS	U. Paris VI – France	Rapporteur
MR THIERRY GAUTIER	INRIA Rhône-Alpes – France	Examinateur
MR AXEL KRINGS	U. of Idaho – USA	Examinateur
MR JEAN LOUIS ROCH	INPG – France	Co-Directeur de thèse
MR DENIS TRYSTRAM	INPG – France	Directeur de thèse

À la mémoire de mon père ...

À ma moitié : Izdihar ...

Aux lumières de ma vie : Kinan et Rayan ...

Remerciements

Avant tout, je voudrais exprimer ma fierté de mon pays la Syrie qui a fait beaucoup d'efforts pour offrir à ses enfants les moyens d'étudier à l'étranger pour qu'ils puissent participer activement à son développement. Je tiens également à remercier, mon université d'origine et future, l'Université de Damas.

À l'époque, j'étais assistant à l'Université de Damas et avec beaucoup de chance j'ai eu l'opportunité de rencontrer le professeur Xavier Rousset de Pina à Damas qui m'a sélectionné et m'a encouragé à poursuivre mes études supérieurs en France. Mes premiers remerciements vont naturellement à toi Xavier, Merci pour tout ...

Mes respects et mes remerciements à mon jury. Je tiens à remercier Van-Dat Cung pour m'avoir fait l'immense honneur de présider le jury de ma soutenance. Merci à Mohamed Jemni et Pierre Sens pour avoir bien voulu consacrer une part de leur temps à rapporter sur ces travaux. Merci à Jean-Louis Roch et Denis Trystram, mes deux directeurs pour leur accueil et leur direction scientifique et humaine. Merci à Axel Krings, avec qui j'ai eu l'immense plaisir d'interagir et de collaborer. Mille mercis à Thierry Gautier, le responsable de l'intergiciel KAAPI pour toutes les aides qu'il a apporté à ce travail, pour sa confiance en moi, merci à toi Thierry pour ces années magnifiques qu'on a passé ensemble.

Je tiens à remercier tous les membres du laboratoire ID-IMAG (MOAIS et aussi MESCAL), sans citation de leurs noms pour ne pas en oublier, pour leur gentillesse et leur accueil.

Enfin, merci à celle qui me donne l'envie, la joie et le soutient d'évoluer. Merci à toi Izdihar pour tout ... Kinan et Rayan seront fières de nous pour tout ce qu'on fait ensemble pour eux.

Table des matières

Table des figures

Chapitre 1

Introduction

Sommaire

1.1 Introduction

Les progrès remarquables des équipements informatiques et de télécommunications durant ces dernières années ont permis une forte évolution des environnements répartis et parallèles qui les utilisent. On est ainsi passé de réseaux locaux de stations de travail à des réseaux à grande échelle de machines. Cette avancée des équipements a permis l'apparition de nouvelles architectures parallèles de grande taille comme les grappes, les grilles et les systèmes pair à pair. Ces architectures, différentes des architectures symétriques à mémoire partagée SMP (*Symetric Multi-Processors*), sont conçues pour répondre plus efficacement aux besoins des acteurs des différents domaines scientifiques. Ces besoins se caractérisent par des programmes demandant de plus en plus de puissance de calcul tant en temps processeur qu'en espace mémoire. L'évolution des moyens disponibles et des besoins exprimés par les utilisateurs sur les environnements de calcul parallèle peut être résumée comme suit :

- Applications parallèles réparties : les besoins en puissance de calcul, ont accru de manière très importante le développement d'applications parallèles. L'utilisation du parallélisme permet, par exemple, de simuler certains phénomènes ne pouvant pas être expérimentés réellement [23, 66] ou d'élargir le domaine des problèmes d'optimisation combinatoire pouvant être résolus [2] ;
- Hétérogénéité : les composants utilisés par les applications parallèles développées, que ce soient les machines ou les infrastructures de communication qu'elles utilisent, se sont diversifiés. Les machines peuvent être des machines SMP, des stations de travail ou même des machines portables. Les réseaux de communication utilisés pour les interconnecter peuvent être, quant à eux, des réseaux locaux (Ethernet, Myrinet ou Infini Band) ou des réseaux à grande échelle (Internet) ;
- Dynamicité : le nombre de ressources dans les infrastructures de calcul parallèle n'est pas fixé. En effet, le nombre de ressources disponibles peut varier à tout moment et cela même durant l'exécution d'une application.

La prise en compte de ces évolutions impose de nouvelles contraintes au niveau des systèmes d'exploitation et/ou des outils qui permettent d'implanter les applications parallèles :

- Prise en compte des pannes : puisque les résultats de l'exécution répartie d'une application parallèle dépendent de plusieurs nœuds interconnectés par un réseau, la probabilité de voir une panne intervenir durant l'exécution d'une application sur une architecture parallèle répartie devient forte : Leslie Lamport donne alors la définition suivante d'un système réparti [76] :
  "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable".
  Dans un tel contexte, la sûreté de fonctionnement des applications est un élément de première importance et un mécanisme de tolérance aux pannes devient nécessaire pour en assurer certains aspects.
- Prise en compte de l'hétérogénéité : la diversité des équipements informatiques introduit une forte hétérogénéité matérielle dans les plateformes de calcul global telles que les grilles et les systèmes pair à pair. Ceci a motivé des techniques d'abstraction des ressources de calcul fournissant un état d'exécution indépendant des ressources physiques mises en œuvre. Ce résultat est obtenu soit par l'intermédiaire de la définition d'une machine virtuelle (Java [82], Python [1]), soit par la description formelle de l'état

[1] http ://www.python.org/

d'exécution de l'application [115, 49].
- Prise en compte de la dynamicité : le nombre de ressources disponibles sur la plateforme de calcul global varie d'un instant à l'autre. Cela provient de la capacité de ces plateformes à recevoir de nouvelles ressources ou à supprimer une ou plusieurs des ressources utilisées en cas de besoin (panne, maintenance, ou utilisation plus prioritaire). L'application parallèle doit donc être capable de s'adapter pendant toute son exécution au nombre de ressources dont elle dispose effectivement sur la plateforme de calcul.

Introduire des solutions aux problèmes précédents dans des outils de programmation parallèle et/ou des systèmes d'exploitation a conduit à intensifier les travaux de recherche afin de disposer d'une exploitation facile, fiable et efficace des environnements de calculs parallèles. Ces travaux portent notamment sur la sûreté de fonctionnement d'un système informatique par la tolérance aux pannes [5, 78, 45].

Bien que plusieurs solutions soient proposées dans la littérature pour traiter les défaillances matérielles ou logicielles [118, 45], l'exploitation efficace et fiable des architectures de grande taille (grappe, grille, pair à pair) reste limitée à cause des difficultés résultant de l'hétérogénéité et de la variabilité du nombre des nœuds utilisés par le calcul sur ces architectures [115, 19]. Cependant, la prise en compte de ces deux facteurs est essentielle si on veut pouvoir tirer pleinement profit de ces nouvelles architectures. C'est ce qui nous a conduit à nous intéresser à ce problème.

1.2 Objectifs

L'objectif de ce travail est l'analyse et le développement d'un mécanisme de tolérance aux pannes pour un système parallèle distribué constitué d'un nombre variable de ressources hétérogènes. Ce système doit permettre à une application de s'exécuter correctement, avec des garanties d'efficacité, sur une architecture dynamique (avec suppression et ajout de ressources). Ce travail de recherche s'insère dans le développement de l'interface de programmation parallèle ATHAPASCAN [49] et de son moteur exécutif distribué KAAPI développés au sein du laboratoire ID-IMAG.

KAAPI signifie *Kernel for Adaptive and Asynchronous Parallel Interface*. Il s'agit d'un intergiciel qui permet la programmation d'une application parallèle et distribuée. Le moteur exécutif KAAPI peut être utilisé comme machine cible pour des langages de plus haut niveau, comme ATHAPASCAN [49] et HOMA [52].

Une application de KAAPI décrit à l'exécution un **graphe de flot de données** qui représente l'enchaînement de tâches qui se synchronisent par accès à des données partagées. La description du parallélisme est indépendante de l'architecture distribuée sous-jacente : un algorithme d'ordonnancement dynamique décide des machines cibles pour l'exécution des tâches et le stockage des données.

L'objectif de notre travail est donc de proposer un mécanisme de tolérance aux pannes, adapté à ATHAPASCAN et à son moteur exécutif KAAPI , permettant :
- l'ajout et la suppression dynamique de ressources ;
- le traitement de la panne d'un ou plusieurs nœuds sans avoir à redémarrer globalement l'application ;
- la reprise sur n'importe quel autre type de nœud disponible d'un ou plusieurs des nœuds défaillants (hétérogénéité) ;

– la minimisation, dans le système, du nombre de composants fiables durant toute l'exécution de l'application ;
– un surcoût de temps d'exécution raisonnable en l'absence de panne et qui reste faible en cas de pannes ;
– l'invisibilité des mécanismes proposés pour l'utilisateur (transparence).

1.3 Contributions

L'étude de la tolérance aux pannes des applications dans les environnements parallèles et répartis n'est pas un problème nouveau [19, 45, 46, 93, 114, 113]. L'originalité de notre travail repose d'une part sur le choix du concept de graphe de flot de données pour représenter de manière distribuée l'état d'exécution d'un calcul global, et d'autre part sur le fait que les mécanismes fournis ne dégradent pas l'efficacité des applications parallèles qui les utilisent.

La représentation de l'état de l'exécution d'un programme parallèle par un graphe de flot de données garantit la description du parallélisme indépendamment du nombre de ressources et peut donc être exploité pour résoudre les problèmes liés à la dynamicité des plateformes considérées. La définition de formats portables pour la représentation des nœuds du graphe résout les problèmes d'hétérogénéité. La sauvegarde du graphe de flot de données d'une application durant son exécution sur une plateforme, constitue un point de reprise pour cette application. Par la suite, une reprise de l'exécution de l'application est possible sur un autre type de plateforme à partir de ce point de reprise.

Le mécanisme de tolérance aux pannes que nous proposons repose sur la sauvegarde du graphe de flot de données d'une application parallèle sur un support stable de stockage. Nous proposons deux protocoles de sauvegarde l'un systématique [63, 64] indépendant de l'ordonnancement et l'autre utilisant un ordonnancement de type vol de travail [61].

Une analyse théorique et un modèle de coût algorithmique de chacun des deux protocoles proposés sont étudiés [59, 60]. Leur validation expérimentale est effectuée sur des applications développées en ATHAPASCAN [53] et KAAPI [65]. En particulier, nous l'avons appliqué à des applications d'optimisation combinatoire dans le cadre de l'ACI Grille DOCG (Défis en Optimisation Combinatoire sur Grille) en collaboration avec le laboratoire PRISM à Versailles, Gilco à Grenoble et LIFL à Lille [41].

Nous proposons également une étude sur l'utilisation de la sauvegarde et de la reprise du graphe de flot de données comme une solution possible pour l'adaptation de l'exécution d'une application parallèle, en fonction des ressources disponibles sur la plateforme ou du comportement de l'algorithme exécuté. L'objectif est de garantir l'efficacité et les performances de l'application pendant son exécution en fonction des ressources disponibles à chaque instant [62].

Nous proposons, enfin, l'utilisation de la technique de sauvegarde / reprise du graphe de flot de données pour certifier l'exécution d'une application parallèle, répartie [64, 71, 70] sur une architecture de grande taille, pouvant potentiellement être victime d'attaques massives (chevaux de troie, virus, etc).

1.4 Cadre de travail

Ces travaux de recherche ont été effectués au laboratoire ID-IMAG, au sein du projet MOAIS (Multi-programmation et Ordonnancement sur ressources distribuées pour les Applications Interactives de Simulation) [90], un projet commun CNRS, INRIA, INPG, UJF. MOAIS étudie la programmation des applications dans lesquelles l'exploitation d'un nombre croissant de ressources est une clef pour améliorer la performance. La construction d'algorithmes parallèles répartis adaptatifs est l'un des axes de recherche de MOAIS.

1.5 Organisation du manuscrit

La suite de ce document est organisée en trois parties : la **première partie**, incluant les chapitres 2 et 3, se concentre sur le positionnement du problème de la sûreté de fonctionnement.

- Le chapitre 2 présente les motivations et le vocabulaire relatif à la sûreté de fonctionnement ; il introduit la tolérance aux pannes comme un moyen de sûreté de fonctionnement. Les deux principales techniques de tolérance aux pannes y sont présentées. Le choix d'une technique adaptée à notre contexte termine ce chapitre.
- Les techniques de tolérance aux pannes basées sur le concept de mémoire stable sont présentées dans le chapitre 3 qui se divise en deux parties. La première partie décrit les différents protocoles proposés dans la littérature pour les techniques s'appuyant sur une mémoire stable. La seconde partie présente des critères de comparaison et de classification des systèmes tolérants aux pannes utilisant une mémoire stable. Ces critères sont ensuite utilisés pour présenter les systèmes existants et pour concevoir notre propre mécanisme de tolérance aux pannes.

La **seconde partie** (chapitres 4, 5, 6 et 7) est une présentation du modèle d'exécution et de nos protocoles de sauvegarde/reprise associés.

- Le modèle de programmation et d'exécution cible de ce travail ainsi que l'intergiciel KAAPI qui constitue le cadre précis de ce travail sont présentés dans le chapitre 4.
- Le chapitre 5 présente les deux protocoles de tolérance aux pannes appelés SEL (*Systimatic Event Loging*) et TIC (*Theft-Induced Checkpointing*). L'analyse théorique de coût et la gestion de la tolérance aux pannes sont également présentées.
- Des expérimentations élémentaires et les comparaisons avec le système Satin [123] des deux protocoles proposés font l'objet du chapitre 6.
- La validation et les mesures de performances de nos protocoles de tolérance aux pannes sur une application d'optimisation combinatoire dans le cadre de l'ACI Grille DOCG et du projet IMAG-INRIA AHA sont présentées dans le chapitre 7.

La **troisième partie** (chapitres 8 et 9) présente l'application de nos protocoles de tolérance aux pannes.

- Le chapitre 8 présente une approche pour l'adaptabilité des applications parallèles réparties sur les plateformes de calcul utilisées avec nos protocoles de tolérance aux pannes.
- Le chapitre 9 présente l'utilisation du protocole SEL comme base fondamentale pour détecter certains types de pannes, comme les pannes par valeur.

– Enfin, le chapitre 10 conclut ce travail ; après avoir rappelé le problème posé et les solutions que nous proposons, il résume les apports essentiels de ce travail et ouvre quelques nouveaux éléments de réflexion et perspectives de recherche.

Première partie

Tolérance aux pannes : État de l'art

Chapitre 2

Tolérance aux pannes & Sûreté de fonctionnement

Sommaire

2.1 Introduction

L'objectif de ce chapitre est de présenter le vocabulaire relatif à la "sûreté de fonctionnement" et d'introduire la tolérance aux pannes comme un moyen de la sûreté de fonctionnement. Nous y présentons les deux principales techniques utilisées pour réaliser la tolérance aux pannes. Enfin, dans la dernière section, nous précisons l'approche choisie dans le contexte de notre travail ainsi que les hypothèses sous-jascentes.

2.2 Sûreté de fonctionnement

La sûreté de fonctionnement d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qui leur est délivré [5].

Selon les applications cibles du système informatique, les différentes facettes de la sûreté de fonctionnement se voient accorder une importance plus ou moins grande. La sûreté de fonctionnement peut être considérée selon des points de vue différents mais complémentaires comme le montre Laprie dans [78]. Dans cette étude, pour la sûreté de fonctionnement d'un système informatique, Laprie propose trois points de vue présentés dans les sous-sections suivantes : les entraves à la sûreté de fonctionnement, les attributs de la sûreté de fonctionnement et les moyens permettant de l'assurer.

2.2.1 Entraves à la sûreté de fonctionnement

Les fautes, les erreurs et les défaillances sont les causes et les conséquences de la non-sûreté de fonctionnement [5].

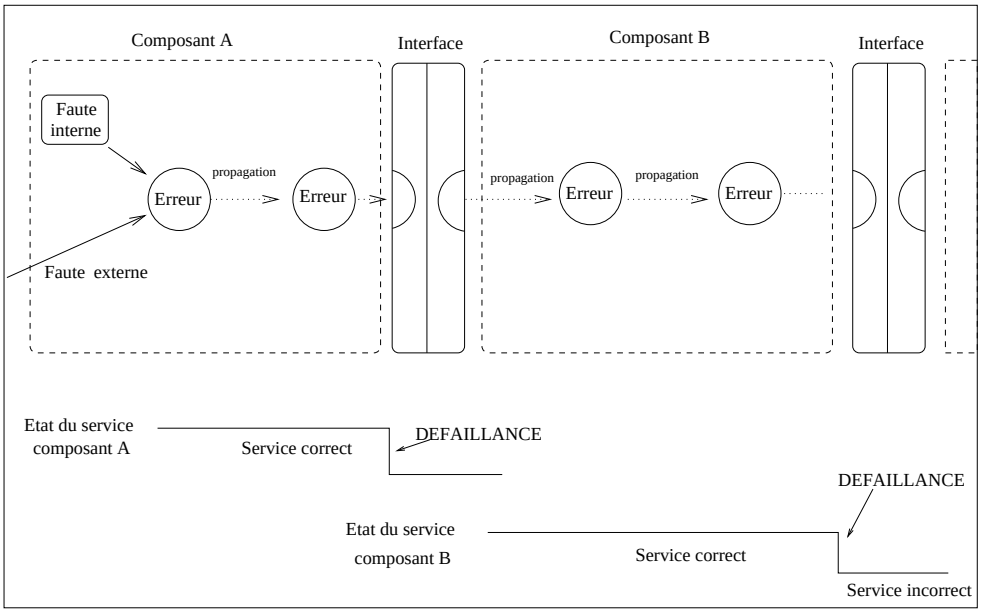

Figure 2.1 – Propagation d'une erreur [5]

Une défaillance du système survient lorsque le service délivré diffère du service spécifié. Comme la figure 2.1 l'illustre, la défaillance survient parce que le système a un comporte-

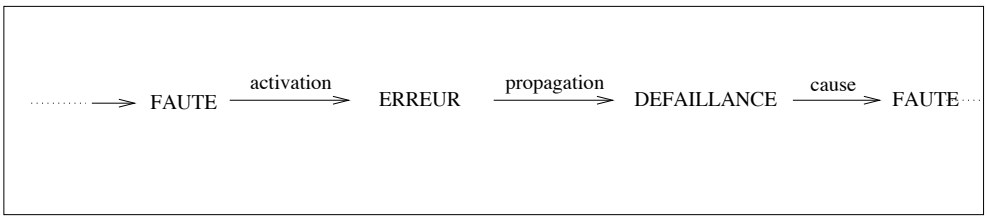

Figure 2.2 – La chaine fondamentale des entraves [5]

ment erroné : une erreur est la partie de l'état du système qui est susceptible d'entraîner une défaillance. La cause adjugée ou supposée de l'erreur est une faute. Une erreur est donc la manifestation d'une faute dans le système, et une défaillance est l'effet d'une erreur sur le service. Ceci conduit à la chaîne fondamentale présentée dans le figure 2.2 :

- **Fautes** : une faute ou panne est caractérisée par sa nature, son origine et son étendue temporelle [78]. La nature d'une faute précise la manière dont elle a été provoquée : intentionnellement ou accidentellement. La cause de l'apparition d'une faute est révélée par son origine. L'étendue temporelle caractérise la persistance ou la durée d'une faute (temporaire ou permanente).
- **Erreurs** : une erreur est la conséquence d'une faute. Une défaillance survient dès que le système utilise un état erroné [37]. Dans [98], une classification des erreurs selon leurs types a été proposée. Premièrement, le service ne correspond pas en valeur à celui spécifié. Deuxièmement, le service n'est pas délivré dans l'intervalle de temps spécifié. Plusieurs catégories ont été définies dans [98], le service rendu est toujours en avance ou toujours en retard ou encore arbitrairement en avance ou en retard. Le fait que le système omette de rendre le service est considéré également comme un type d'erreur. Un arrêt (crash) est défini par le fait que le système omet définitivement de délivrer des services.
- **Défaillances** : une défaillance dénote l'incapacité d'un élément du système à assurer le service spécifié par l'utilisateur. La défaillance est caractérisée par son domaine, sa perception par les utilisateurs et ses conséquences sur l'environnement [78].

2.2.2 Attributs de la sûreté de fonctionnement

Les attributs de la sûreté de fonctionnement d'un système mettent plus ou moins l'accent sur les propriétés que doivent vérifier la sûreté de fonctionnement du système. Ces attributs permettent d'évaluer la qualité du service fourni par un système. Dans [78], six attributs de la sûreté de fonctionnement sont définis :

- **disponibilité** : c'est la propriété requise par la plupart des systèmes sûrs de fonctionnement. Il s'agit de la fraction de temps durant laquelle le système est disponible pour des fins utiles (en excluant les temps de défaillance et de réparation) ;
- **fiabilité** : cet attribut évalue la continuité du service i.e. le taux en temps de fonctionnement pendant lequel le système ne subit aucune faute ;
- **sécurité-innocuité** : similaire à la fiabilité mais par rapport aux conséquences catastrophiques causées par les fautes ;
- **confidentialité** : cet attribut évalue la capacité du système à fonctionner en dépit de fautes intentionnelles et d'intrusions illégales ;

- **intégrité** : l'intégrité d'un système définit son aptitude à assurer des altérations approuvées des données ;
- **maintenabilité** : cette propriété décrit la souplesse du système vis-à-vis des modifications apportées en vue de sa maintenance.

L'importance des attributs de la sûreté de fonctionnement présentés ci-dessus est principalement liée aux applications et à leurs besoins. Par exemple, pour les applications critiques comme le pilotage de fusées, une grande importance doit être donnée à tous les attributs de la sûreté de fonctionnement. Les applications parallèles à longue durée d'exécution font prévaloir la maintenabilité et la fiabilité.

2.2.3 Moyens d'assurer la sûreté de fonctionnement

Les moyens utilisés pour assurer la sûreté de fonctionnement sont définis par les méthodes et les approches utilisées pour assurer cette propriété [78]. Les approches les plus connues sont :

la prévention des fautes qui s'attache aux moyens permettant d'éviter l'occurence de fautes dans le système. Ce sont généralement les approches de vérification des modèles conceptuels ;

l'élimination des fautes qui se focalise sur les techniques permettant de réduire la présence de fautes ou leurs impacts. Cela est réalisé par des méthodes statiques de preuve de la validité du système (simulation, preuves analytiques, tests, ...) ;

la prévision des fautes qui prédit l'occurence des fautes (temps, nombre, impact) et leurs conséquences. Ceci est réalisé généralement par des méthodes d'injection de fautes afin de valider le système relativement à ces fautes ;

la tolérance aux pannes ou aux fautes [1] qui essaye de fonctionner en dépit des fautes. Le degré de tolérance aux pannes se mesure par la capacité du système à continuer à délivrer son service en présence de fautes.

La figure 2.3 résume les notions associées à la sûreté de fonctionnement présentées ci-dessus.

2.3 Classes de pannes

La capacité d'identification du modèle de pannes joue un rôle très important si l'on veut réaliser la tolérance aux pannes. En effet, les conséquences et le traitement d'une panne diffèrent selon son modèle.

Les pannes peuvent être classées selon plusieurs critères : leur degré de gravité, leur degré de permanence et leur nature.

[1] Nous utilisons le mot panne qui veut dire faute, car c'est le terme panne qui est généralement utilisé en français.

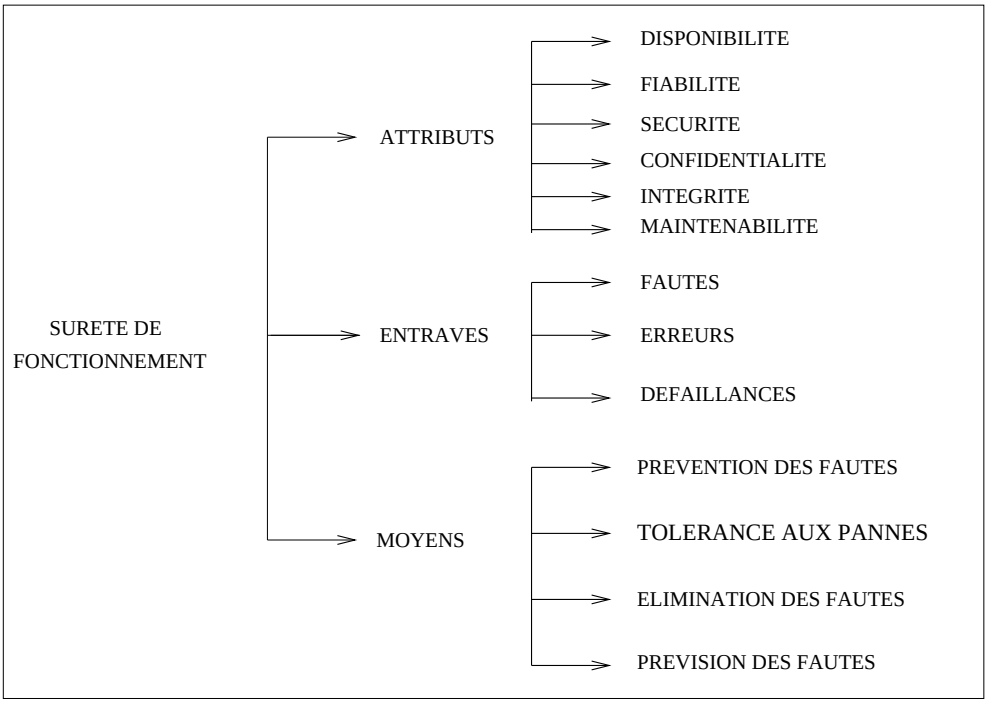

Figure 2.3 – L'arbre de la sûreté de fonctionnement [78]

Un premier classement des pannes selon le degré de gravité des défaillances [5] conduit à :

- **pannes franches (crash fault)** : soit le système fonctionne normalement (les résultats sont corrects), soit il ne fait rien [108, 99]. Il s'agit du modèle de panne le plus simple auquel on essaie de se ramener chaque fois que cela est possible ;
- **pannes par omission** : des messages sont perdus en entrée et/ou en sortie [97]. Ce type de panne est utilisé pour les réseaux ;
- **pannes de temporisation** : les déviations par rapport aux spécifications concernent uniquement le temps (typiquement le temps de réaction à un événement) [39] ;
- **pannes par valeurs** : les résultats produits par un composant défaillant sont incorrects [79] ;
- **pannes byzantines** : le système peut faire n'importe quoi, y compris avoir un comportement malveillant [109, 77].

Un second classement des pannes selon leur degré de permanence [5] est également possible :

- **panne transitoire** : elle se produit de manière isolée ;
- **panne intermittente** : elle se produit aléatoirement plusieurs fois ;
- **panne permanente** : elle persiste dès qu'elle apparaît jusqu'à réparation.

On utilise également un troisième classement selon la nature de la panne [5] :

- **pannes accidentelles** : elles se produisent de manière accidentelles ;
- **pannes intentionnelles** : elles sont créées avec une intention qui peut être malicieuse.

Dans le cadre de cette thèse, on s'intéresse prioritairement aux traitements associés aux pannes franches (*crash fault*) des processus. Plus précisement, on ne traite que les pannes qui peuvent être modélisées et considérées comme des pannes franches. Typiquement, la

volatilité des machines sur une plateforme dynamique peut être considérée comme une panne franche. Cependant, on s'intéresse aussi aux pannes intentionnelles dans le chapitre 9.

2.4 Non-Fiabilité dans les architectures de grande taille

L'exécution de programmes parallèles extrêmement coûteux en calcul comme des applications d'optimisation combinatoire nécessite un grand nombre de processeurs pour résoudre le problème dans un temps raisonable. Par exemple pour certaines instances de ces problèmes, l'exécution nécessite environ 1000 processeurs pendant 7 jours [2].

Aussi, les plateformes actuelles de calcul contiennent des milliers de processeurs. Ainsi, la plateforme Grid5000 est une grille expérimentale d'environ 1496 CPUs constituée d'un ensemble de grappes géographiquement réparties sur l'ensemble du territoire français (Bordeaux, Grenoble, Lille, Lyon, Orsay, Rennes, Nancy, Sophia-Antipolis et Toulouse) et interconnectées par le réseau haut débit Renater.

L'utilisation d'un tel nombre de processeurs augmente de manière considérable la probabilité de voir des pannes intervenir durant l'exécution [119].

Considérons un système composé de n processeurs. Par définition la **fiabilité** $R(t)$ de ce système est la probabilité qu'il soit opérationel pour tout instant $t_i \in [0, t]$.

Les auteurs de [40, 112] montrent que $R(t) = e^{-\lambda tn}$, où λ est une constante représentant le taux de défaillance sur un processeur. La constante λ est l'inverse du temps moyen entre deux pannes, notée MTBF (Mean Time Between Failures) du système : $\lambda = \frac{1}{MTBF}$.

La probabilité $F(t)$ que le système ne soit pas opérationel est la **non fiabilité** du système sur l'intervalle de temps $[0, t]$; elle s'écrit [112] :

$$F(t) = 1 - R(t) = 1 - e^{-\lambda tn} \tag{2.1}$$

Ainsi, comme la figure 2.4 l'illustre, la probabilité de défaillance des applications parallèles réparties augmente considérablement en fonction de la durée d'exécution et du nombre de processeurs utilisés.

2.5 Tolérance aux pannes dans les systèmes répartis

Afin de présenter les techniques de tolérance aux pannes dans les systèmes répartis, nous commençons cette section par une définition simple d'un système réparti.

2.5.1 Systèmes répartis

De manière générale [38, 117], un système réparti peut être défini comme un ensemble de nœuds de calcul reliés par un réseau de communication et communiquant entre eux par messages.

Dans ce contexte, la tolérance aux pannes est un besoin imposé par la répartition et c'est pourquoi plusieurs travaux de recherches sur les systèmes répartis ont été réalisés afin de la prendre en compte.

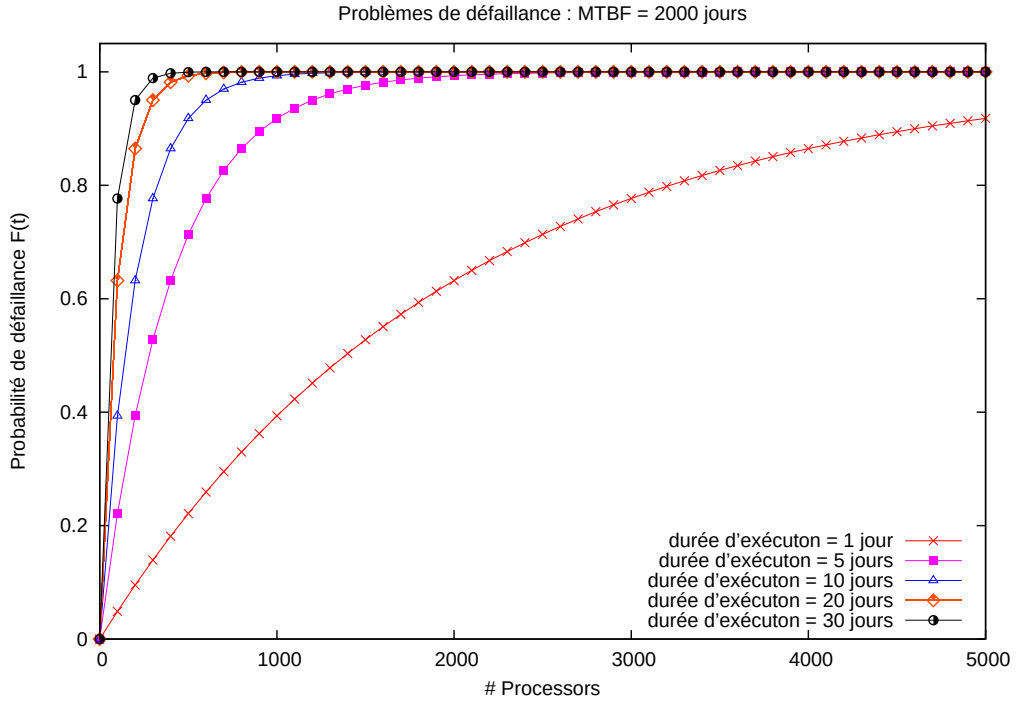

Figure 2.4 – La probabilité de défaillance dans les environnements parallèles répartis pour différentes durées d'exécution, dans le cas où $MTBF = 2000$ jours ~ 6 ans pour chaque processeur (i.e. $\lambda = 0.0005$)

2.5.2 Techniques réparties de tolérance aux pannes

La tolérance aux pannes dans les systèmes répartis et parallèles est toujours réalisée par l'emploi d'un mécanisme de redondance [3]. Cette dernière peut être spatiale (duplication de composants), temporelle (traitements multiples) ou bien informationnelle (redondance de données, codes, signatures). Dans un tel système à base de processus communicants, les techniques de tolérance aux pannes peuvent être séparées en deux classes : les techniques basées sur la duplication [110, 24] et les techniques basées sur une mémoire stable [100, 104].

2.5.2.1 Tolérance aux pannes par duplication

La tolérance aux pannes par duplication consiste en la création de copies multiples des composants sur des processeurs différents. Cette approche par duplication rend possible le traitement des pannes en les masquant. Trois stratégies principales pour réaliser la duplication sont proposées dans la littérature : les duplications actives, passives et semi-actives. Ces différentes stratégies visent à garantir une cohérence forte entre les copies d'un composant dupliqué [126].

- **Duplication active** : elle est définie par la symétrie du comportement des copies d'un composant dupliqué où chaque copie joue un rôle identique à celui des autres.
- **Duplication passive** : contrairement à la duplication active, la duplication passive (*passive replication ou primary-backups replication*) [89] est asymétrique. Elle dis-

tingue deux comportements pour les copies d'un composant dupliqué : la copie primaire (primary copy) et les copies secondaires (backups). La copie primaire est la seule à effectuer tous les traitements. Les copies secondaires, oisives, surveillent la copie primaire. En cas de défaillance de la copie primaire, une des copies secondaires devient la nouvelle copie primaire.

– **Duplication semi-active** : la duplication semi-active (semi-active replication ou leader-followers replication) [89] se situe à mi-chemin entre la duplication active et la duplication passive. Comme cette dernière, la duplication semi-active est une stratégie asymétrique. Contrairement à la duplication passive, les copies secondaires ne sont pas oisives.

La tolérance aux pannes par duplication est alors réalisée par masquage d'erreur. La défaillance d'une copie est masquée par le comportement des copies non défaillantes. Cette technique permet de tolérer un nombre limité de pannes (au maximum $\#duplicata - 1$ dans le cas de pannes franches) et n'est pas adaptée aux calculs parallèles massifs en terme de performances et de ressources de calculs nécessaires à sa mise en œuvre.

2.5.2.2 Tolérance aux pannes par mémoire stable

L'existence d'une mémoire stable permet la réalisation de la tolérance aux pannes par une détection de défaillance suivie d'un recouvrement d'erreur.

Mémoire stable La mémoire stable n'est qu'une abstraction [45]. Elle peut être définie comme un support persistant de stockage, dont le rôle principal est d'assurer une accessibilité et une protection aux données contre les pannes pouvant affecter le système. Ainsi, suite à une panne, un état correct ayant été stocké antérieurement à cette panne sur la mémoire stable reste accessible ; cela permet au système un retour à un état anterieur.

Un support de stockage est vu comme une mémoire stable si et seulement si les trois conditions suivantes sont vraies :

1. **Accessibilité :** il existe à tout moment de l'exécution un chemin permettant d'accéder aux données sauvegardées même en présence des pannes.
2. **Protection :** les pannes affectant le système ou l'application n'altèrent pas les données.
3. **Atomicité :** les mises à jour des données sur le support de stockage se font de manière atomique.

Souvent, il y a une confusion entre la mémoire stable et le composant matériel qui l'implante. La mémoire stable doit garantir la persistance et l'accessibilité des données nécessaires à une reprise après panne. Ceci peut conduire à différentes mises en œuvre d'une mémoire stable :

– dans les approches où le mécanisme de tolérance aux pannes ne tolère qu'une seule panne, la mémoire stable d'un processus peut correspondre à la mémoire volatile ou au disque d'un autre processus [18, 45] ;
– si l'objectif de la tolérance aux pannes est de tolérer un nombre arbitraire de pannes transitoires, le disque dur local de chaque processus peut être utilisé pour implanter sa mémoire stable ;
– pour un système qui tolère les pannes permanentes (non transitoires), la mémoire stable est un support de stockage persistant situé en dehors des nœuds exécutant les processus de calcul ; celle-ci reste accessible à tout moment [45].

Principe de la tolérance aux pannes par mémoire stable Le principe de cette technique est de remplacer l'état d'erreur par un état correct en se basant sur l'abstraction d'une mémoire stable présentée dans le paragraphe ci-dessus. Ceci nécessite tout d'abord la détection de l'erreur, c'est à dire, identifier la partie incorrecte de l'état, afin de pouvoir ensuite effectuer le recouvrement d'erreur.

L'identification de la partie incorrecte de l'état d'un système (détection de défaillances), se fait généralement au moyen d'un test de vraisemblance. Ce dernier peut être **explicite** en vérifiant des propriétés spécifiques de l'état, ou **implicite** via des conséquences visibles comme par exemple la violation d'un délai de garde.

Les objectifs de la détection d'erreur sont de prévenir, si possible, l'occurrence d'une défaillance provoquée par l'erreur, d'éviter la propagation de l'erreur à d'autres composants (voir figure 2.1) et de faciliter l'identification ultérieure de la faute en vue de son élimination ou de sa prévention. Bien que, la détection de défaillance soit une partie très importante des techniques de tolérance aux pannes par mémoire stable, elle ne fait pas partie du travail de recherche décrit dans cette thèse. Pour plus de détails sur cet axe actif de recherche nous invitons le lecteur à regarder les travaux de Chandra et Toueg [33], de Schiper [106, 107, 111], de Sens [11, 87] et ceux de Hurfin et Raynal [58].

Une fois qu'une panne est détectée, un mécanisme de recouvrement doit être mis en place pour traiter la panne identifiée. Il existe deux techniques de recouvrement [78] :

- **la reprise :** c'est la technique générale qui consiste à retourner vers un état antérieur dont on sait qu'il est correct. Cette technique nécessite donc la sauvegarde régulière de l'état du système ou de l'application.
- **la poursuite :** c'est une technique spécifique qui consiste à reconstituer un état correct, sans retour en arrière. La reconstitution n'est souvent que partielle, d'où un service dégradé.

Comme nous l'avons déjà remarqué, pour pouvoir réaliser une reprise après une panne, l'état du système ou de l'application doit être sauvegardé périodiquement ou lors de l'occurence d'événements spécifiques. L'état sauvegardé doit constituer un point de reprise à partir duquel le système peut fonctionner correctement. Pour assurer cette propriété, le mécanisme de tolérance aux pannes doit résoudre les problèmes de reprise suivants :

- La sauvegarde et la restauration d'un point de reprise doivent être atomiques. Une action atomique est une opération O qui fait passer en un temps fini un ensemble d'objets E_O d'un état initial à un état final. Une action atomique est caractérisée par les propriétés suivantes :
  1. L'état de E_O n'est pas accessible aux autres actions pendant l'exécution de O ;
  2. Si une défaillance survient durant l'exécution de O, E_O se trouve dans son état initial ;
- L'état des points de reprise doit lui-même être protégé contre les fautes ;
- Dans un système réparti, les points de reprise sont créés pour chaque processus. L'ensemble de ces points de reprise doit constituer un état global cohérent du système [34].

Le point critique pour réaliser la tolérance aux pannes par mémoire stable dans les systèmes répartis est la constitution d'un état de reprise cohérent qui réponde aux objectifs attendus de la tolérance aux pannes : QoS, surcoût introduit, type de pannes à traiter, en cas de défaillance reprise globale du système ou bien uniquement reprise des processus défaillants, nombre de composants fiables dans le système durant toute son exécution, etc... .

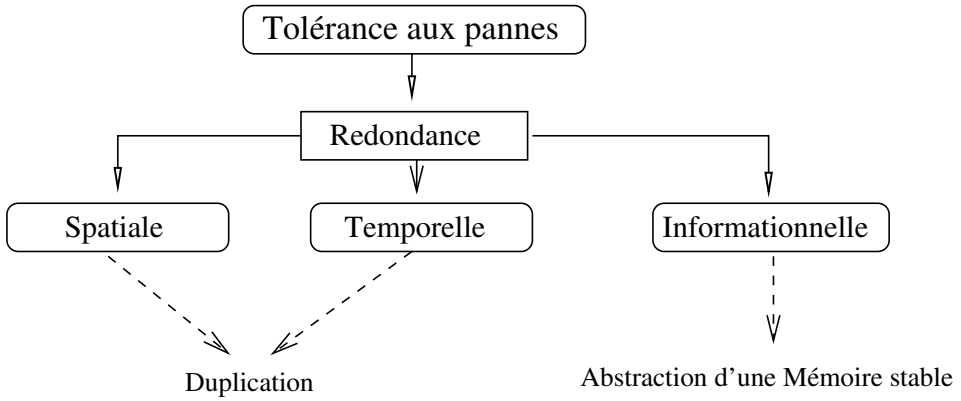

Figure 2.5 – Les techniques de tolérance aux pannes dans les systèmes répartis

Deux approches ont été proposées dans la littérature pour construire un état global cohérent d'un système réparti [45, 34] :

- **A priori** : les sauvegardes des différents états des processus coopérants sont coordonnées pour constituer un état global cohérent ;
- **A posteriori** : les sauvegardes des états des processus sont indépendantes ; la reconstitution d'un état global cohérent se fait lors de la reprise.

Il est à noter que la sauvegarde d'une données sur une mémoire stable peut être implantée par des techniques de codage pour limiter la redondance. Parmi les systèmes utilisants cette technique, on peut citer les systèmes RAID (*Redundant Array of Independent Disks*), les serveurs de stockage paire à paire. Cette technique a été proposée pour les systèmes de calcul parallèle [36].

2.6 Bilan et Conclusion

Dans ce chapitre, nous avons présenté la notion de sûreté de fonctionnement d'un système informatique. Nous avons vu que la tolérance aux pannes n'est qu'un moyen pour assurer la sûreté de fonctionnement d'un système et qu'on utilise toujours la redondance pour l'obtenir. Deux méthodes principales permettent de réaliser la tolérance aux pannes dans les systèmes répartis (figure 2.5) : la duplication et l'abstraction d'une mémoire stable.

Dans les systèmes parallèles où la durée du calcul est très importante, toutes les ressources disponibles doivent être utilisées pour le calcul lui-même. La tolérance aux pannes par duplication n'est donc pas adaptée compte tenu de la quantité de ressources nécessaires à la duplication du traitement. A cela, il faut ajouter le sûrcout important lié à la gestion des copies et la limitation à un nombre fixé à l'avance du nombre de pannes recouvrables. Par conséquent, la tolérance aux pannes par mémoire stable semble la méthode la plus adaptée aux systèmes parallèles. Ce qui nous conduit à adopter le principe d'une mémoire stable pour réaliser la tolérance aux pannes franches pour des applications parallèles s'exécutant sur une plateforme dynamique.

Avec cette orientation, nous présentons dans le chapitre suivant, les différents problèmes liés à la tolérance aux pannes par reprise et les diverses solutions proposées. Ensuite, nous définissons quelques propriétés pour la tolérance aux pannes par reprise. Enfin, nous dé-

crivons et analysons quelques systèmes parallèles tolérant les pannes par reprise selon ces propriétés.

Pour conclure ce chapitre, il faut noter qu'il n'existe pas de méthode de tolérance aux pannes qui soit valable dans l'absolu [4]. Seules existent des méthodes adaptées à des hypothèses particulières d'occurrence de fautes.

Chapitre 3

Tolérance aux pannes par reprise & Applications parallèles

Sommaire

3.1 Introduction

Comme nous l'avons vu dans le chapitre précédent, la tolérance aux pannes par reprise (*rollback*) est basée sur la redondance des informations, autrement dit, sur la sauvegarde de données à partir desquelles une reprise du calcul est possible en cas de panne. Ceci nécessite d'une part, l'existence d'une mémoire stable de sorte que les données soient toujours accessibles et, d'autre part, que l'état du système ou du calcul soit cohérent après une reprise.

La condition générale pour qu'une reprise après panne soit correcte peut être définie comme suit : **un système (application) peut réaliser une reprise correcte si son état interne est cohérent avec un comportement du système observé avant la panne** [114]. Par conséquent, le protocole de reprise doit essentiellement maintenir les informations concernant les interactions non seulement entre les processus participants, mais également les interactions, s'il y en a, avec le monde extérieur.

3.2 Application parallèle

Afin de définir une application parallèle, nous introduisons les définitions suivantes :

Définition 1 *Une **tâche** est une séquence d'instructions exécutée localement sur une unique ressource, éventuellement de manière non préemptive.*

Définition 2 *Une **application parallèle** est un ensemble de tâches éventuellement soumises à des conditions de synchronisation pouvant être potentiellement exécutées en même temps et sur des sites géographiques différents. L'objectif principal est de limiter le temps d'exécution de l'application.*

Une application parallèle est donc souvent composée de n processus distincts communiquant via un réseau d'interconnexion, chacun des processus exécutant des tâches et les communications qui leurs sont associées. Par conséquent, l'état de l'application parallèle à un instant t donné, est l'ensemble des états locaux de tous les processus participants et de tous les canaux de communications ; autrement dit, de tous les messages en transit à l'instant t. Cependant, un processus ne peut stocker que son état local et ses messages entrants et sortants. Aussi, les messages en transit à un instant donné, ne peuvent être contrôlés par aucun des processus de l'application parallèle.

De manière générale, la définition 2 d'une application parallèle nous place dans le contexte d'un système réparti. Par conséquent, le raisonnement sur le système ou l'application (état, ordre des événements) pose un certain nombre de problèmes liés à la répartition.

- L'absence d'une **mémoire commune**, alors que la mémoire est le support habituel de l'état d'une application ou d'un système.
- L'absence d'une **horloge commune**, or l'existence de cette horloge permet la définition d'un ordre sur les événements du système.
- L'**asynchronisme des communications** entre les processus induisant que la borne supérieur du temps de transmission d'un message est a priori inconnue.
- L'**hétérogénéité** des sites d'exécution.

En 1978, Lamport a proposé de définir pour les systèmes répartis une notion de temps logique permettant de comparer des événements selon leur ordre d'exécution [75]. Sur un

site, les événements locaux peuvent être ordonnés en se basant sur l'ordre de leur exécution. L'émission d'un message sur le site émetteur précède toujours sa réception sur le site récepteur. Ceci correspond à la notion de précédence causale.

3.3 Les problèmes de la reprise

Deux problèmes interviennent pour effectuer un recouvrement d'erreur par reprise : le déterminisme d'exécution et la cohérence d'état global de l'application parallèle (système réparti).

Le déterminisme de l'exécution d'un programme P est le fait de pouvoir répéter l'exécution de P plusieurs fois et de la même façon. Cette propriété d'exécution permet de déterminer si la reprise peut être faite par une technique de journalisation de l'exécution ou non. En effet, le stockage de l'histoire de l'exécution de P dans une mémoire stable est inutile si la réexécution de P n'est pas équivalente à l'exécution de P.

Le concept d'état global cohérent est introduit par la répartition de l'application. Le principe est simple : un état global est cohérent si l'application peut réellement passer par cet état. Autrement dit, un état global ne peut pas être cohérent si par exemple il contient un message reçu sans que celui-ci n'ait été envoyé.

3.4 Exécution déterministe

Afin d'obtenir une exécution déterministe, un processus peut être modélisé par une séquence d'intervalles d'états [114] chacun débutant par un événement non déterministe pouvant être identifié.

L'exécution durant chaque intervalle d'état est déterministe. Ce concept d'intervalle d'état est appelé "hypothèse de déterminisme par morceaux" (PWD : *Piecewise deterministic assumption*). L'hypothèse PWD permet de capturer suffisamment d'informations concernant les événements non déterministes qui intialisent les intervalles d'état.

3.5 État global cohérent d'une application parallèle

Un **état global** d'une application parallèle est une collection des états locaux de tous les processus participant au calcul (un par processus) et les états des canaux de communications entre les processus.

Donc, un **état global cohérent** d'une application parallèle est simplement l'un des états globaux qui intervient durant une exécution correcte, i.e., sans panne. Autrement dit, dans un état global cohérent, si l'état d'un processus contient la réception d'un message m, alors l'état du processus qui a envoyé m contient son envoi [34, 45].

La figure 3.1, montre un exemple d'état global cohérent C_1, C_2, C_3. Les messages m_2 et m_3 ont éte envoyés mais pas reçus, ce qui est normal durant une exécution correcte (sans panne), car à un instant donné, on peut avoir dans le système des messages envoyés, mais pas encore reçus. Donc, l'état global C_1, C_2, C_3 dans la figure 3.1 représente un état global cohérent.

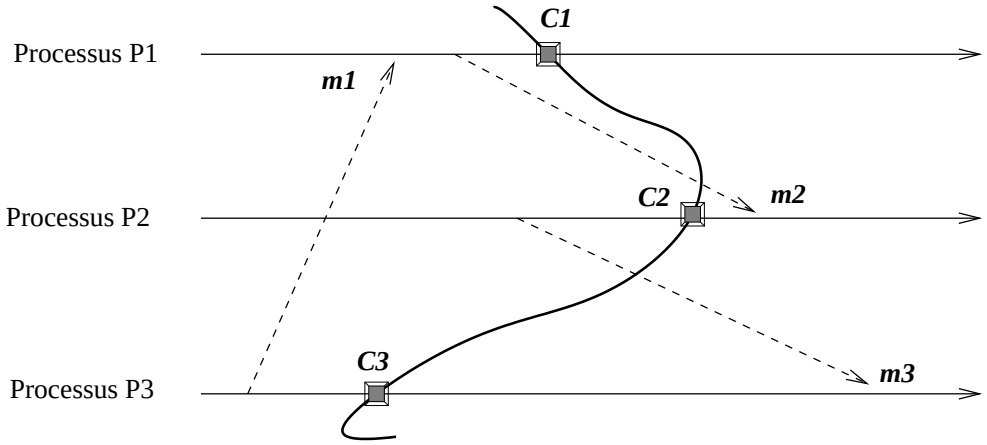

Figure 3.1 – Exemple d'un état global cohérent

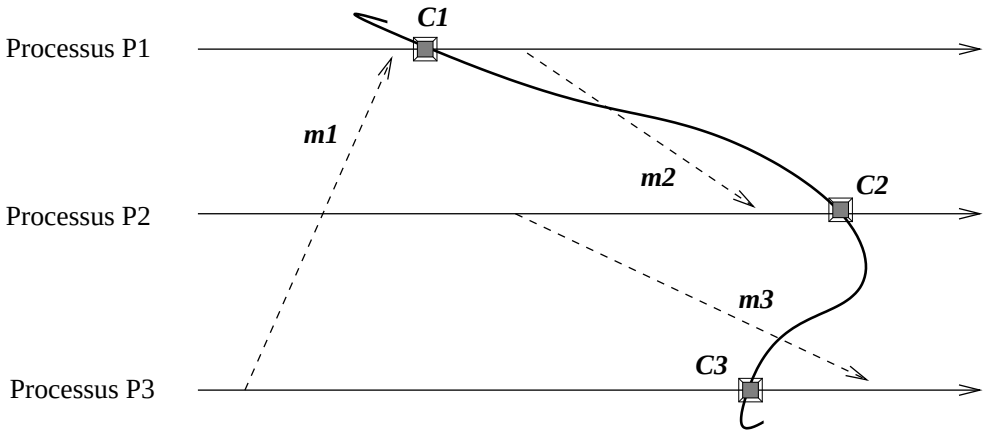

Figure 3.2 – Exemple d'un état global incohérent

Par contre, l'état global C_1, C_2, C_3 illustré dans la figure 3.2, n'est pas cohérent. En effet, le message m_2 n'a pas été envoyé par le processus P_1 alors que m_2 est reçu par le processus P_2. Cette situation, ne peut jamais se produire dans une exécution correcte, c'est pourquoi l'état global présenté dans la figure 3.2 est incohérent.

Nous avons vu dans la figure 3.1 que l'état global C_1, C_2, C_3 contient certains messages (m_2 et m_3) envoyés mais pas reçus. Ces messages sont appelés messages **en transit** et ils font partie de l'état global du système (application). Les messages en transit n'introduisent aucune incohérence dans l'état. Cependant, la garantie de la délivrance des messages en transit doit être assurée, soit par l'hypothèse d'un réseau de communication fiable, soit par le protocole de sauvegarde/reprise lui-même.

Dans un état global, un message reçu sans être envoyé (message m_2 dans le figure 3.2), est appelé message **orphelin**. Un message orphelin cause l'incohérence de l'état global qui le contient.

3.6 Journalisation

Le principe du mécanisme de journalisation (*logging*) est la sauvegarde de l'histoire d'exécution de l'application. Ces mécanismes s'appuient explicitement sur le fait qu'un processus, peut être modélisé par une séquence d'intervalles d'états déterministes (hypothèse PWD) [114]. Chaque intervalle d'état débute par un événement non déterministe, dont l'enregistrement permet la reconstruction de l'état du processus. Un événement non déterministe peut être la réception d'un message ou un événement interne au processus.

La reprise basée sur la journalisation, suppose que tous les événements non déterministes peuvent être identifiés et que les informations correspondant à ces événements peuvent être enregistrées sur la mémoire stable.

En effet, le principe de la reprise basée sur la journalisation est que durant l'exécution normale, c-à-d sans panne (*failure-free execution*), chaque processus stocke dans un journal sur mémoire stable les informations correspondant à tous les événements non déterministes qu'il observe. Après une panne, le processus défaillant reconstruit son état d'avant la panne à partir de son état initial et en utilisant son journal, afin de rejouer les événements non déterministes exactement comme ils se sont produits avant la panne. Pour éviter la reconstruction de l'état d'un processus défaillant à partir de son état initial, des sauvegardes périodiques de l'état du processus sont effectuées.

Traditionnellement, les événements non déterministes sont associés aux messages et le mécanisme de journalisation est donc appelé **journalisation de messages**. Cette association a été proposée et implantée dans [8, 18, 114] où un événement non déterministe est généré sur la réception d'une message.

Les protocoles de reprise basés sur la journalisation doivent assurer la "condition de non orphelinité" suivante : en cas de reprise de tous les processus défaillants, **le système ou l'application ne contient aucun processus orphelin**. Un processus orphelin est un processus dont l'état dépend d'un événement non déterministe, et ce dernier ne peut pas être reproduit par l'opération de reprise. Dans [46], une description formelle de cette condition a été introduite.

Les techniques de reprise par journalisation diffèrent par la façon d'assurer et d'implanter la "condition de non orphelinité" définie précédemment. Nous détaillons par la suite les trois protocoles de journalisation : les journalisations pessimistes, optimistes et causales.

3.6.1 Journalisation pessimiste

La journalisation **pessimiste** [114] se base sur l'hypothèse qu'une panne peut se produire après n'importe quel événement non déterministe dans le système. La conséquence de cette hypothèse "pessimiste" est le stockage sur la mémoire stable de l'information correspondant à chaque événement non déterministe avant qu'il n'affecte le calcul. La propriété de cette technique est que si un événement n'est pas enregistré sur la mémoire stable alors aucun processus ne dépend de cet événement.

Dans le cas où les événements non déterministes sont uniquement des réceptions de messages, aucun processus P ne peut envoyer un message m avant que tous les messages reçus avant l'émission m ne soient journalisés sur la mémoire stable [72].

L'avantage des techniques pessimistes est qu'elles ne créent jamais de processus orphelins. Mais leur inconvenient est qu'elles bloquent le processus avant chaque retrait de mes-

sage et donc la journalisation pessimiste introduit un sûrcout important durant l'exécution normale, c.-à-d., sans panne.

3.6.2 Journalisation optimiste

Afin d'améliorer la performance d'une exécution normale, la journalisation **optimiste** [114] s'appuie sur l'hypothèse "optimiste" qu'un processus peut compléter la journalisation de son événement non déterministe avant qu'une panne ne survienne. Donc, le stockage sur la mémoire stable des informations correspondant à un événement non déterministe est asynchrone et le processus n'a pas besoin de se bloquer pendant le stockage sur la mémoire stable.

Il est évident que, la journalisation optimiste ne garantit pas toujours la "condition de non orphelinité". Ceci rend l'opération de reprise compliquée, car il faut éventuellement retourner en arrière plusieurs fois avant de trouver un état cohérent (à savoir, qui ne contient pas de processus orphelins).

Les inconvénients de cette approche résident dans le risque d'un retour à l'état initial de calcul (pour éviter les processus orphelins), ce phénomène est communément appelé l'**effet domino** [100] . De plus, un sûrcout important dû au calcul de l'état global cohérent est introduit durant la reprise.

3.6.3 Journalisation causale

Le principe de la journalisation causale [44] est d'assurer que le déterminant[1] de chaque événement non déterministe qui précède causalement l'état d'un processus se trouve soit en mémoire stable, soit est disponible localement pour ce processus.

La journalisation causale [45] a l'avantage des deux méthodes précédentes en terme de performances à l'exécution et en cas de reprise. Comme la journalisation optimiste, la journalisation causale évite d'une part, la synchronisation avec la mémoire stable à l'exception du moment de la publication de résultats. D'autre part, comme dans la journalisation pessimiste, la journalisation causale ne crée jamais de processus orphelin. Cela permet la reprise de n'importe quel processus défaillant à partir de son dernier état sauvegardé sans risquer l'effet domino [100]. L'inconvénient de cette technique réside en la complexité du protocole de recouvrement arrière suite à une panne.

La journalisation causale garantit toujours la condition de non orphelinité en évitant les ralentissements dûs à la copie systématique sur mémoire stable des messages reçus. Cela est effectué en contraignant les processus liés par la relation de précédence causale à journaliser les événements non déterministes [72].

L'implantation de la condition de non orphelinité qui est toujours vraie dans la journalisation causale est faite par l'ajout des informations associées à un événement non déterministe (le déterminant d'un événement non déterministe), informations se trouvant dans la mémoire volatile d'un processus, aux messages envoyés vers d'autre processus [44].

[1]Les informations à patir desquelles un événement non déterministe peut être reproduit lors d'une re-exécution

3.7 Sauvegarde

Lors d'une panne, les approches basées sur la sauvegarde d'état des processus participant au calcul restaurent l'état du système ou de l'application à partir du plus récent ensemble cohérent de points de reprise (un point de reprise par processus). Cet ensemble cohérent de points de reprise est appelée **ligne de recouvrement** [100].

Les approches basées sur la sauvegarde (*checkpointing*) ne font pas l'hypothèse PWD sur l'exécution. De plus elles n'ont pas besoin de stocker et de rejouer les événements non déterministes.

La sauvegarde peut être classée selon trois catégories en fonction du mode de construction de la ligne de recouvrement : sauvegarde coordonnée, sauvegarde non-coordonnée et sauvegarde induite par les communications.

3.7.1 Sauvegarde coordonnée

Les protocoles adoptant cette approche sont basés sur les travaux de Chandy et Lamport [34]. Ces algorithmes sont également appelés synchrones, cohérents ou globaux. Le principe de ces algorithmes est la définition de l'état global cohérent (ligne de recouvrement) au moment de la sauvegarde et plus précisément avant de procéder à l'écriture des fichiers de sauvegarde sur la mémoire stable. Ainsi, au moment de la sauvegarde, une phase de synchronisation durant laquelle les calculs sont interrompus est imposée à tous les processus de l'application afin de déterminer l'état global cohérent [96, 116].

Cette approche présente l'avantage de ne pas produire d'effet domino en cas de reprise puisque les points de reprise sont garantis cohérents. De plus, un seul point de reprise par processus est nécessaire ce qui réduit le sûrcout de stockage et de libération de points de reprise.

Le principal inconvénient de cette technique est le surcoût introduit par la coordination de tous les processus participants. Aussi, afin d'améliorer les performances de la sauvegarde coordonnée, plusieurs techniques ont été proposées :

- la sauvegarde coordonnée **non bloquante** [34] évite de bloquer le processus durant la phase de sauvegarde en créant un processus clone chargé de la sauvegarde ;
- la sauvegarde avec **horloge synchronisée** [74] évite la phase de synchronisation par envois de messages, en utilisant une horloge synchronisée ;
- la sauvegarde coordonnée **minimale** [68] évite la phase de synchronisation globale entre tous les processus en se basant sur le fait qu'un processus n'a besoin de se coordonner qu'avec les processus avec lesquels il a des dépendances.

3.7.2 Sauvegarde non coordonnée

Les techniques non coordonnées [45] évitent la phase de synchronisation en laissant à chaque processus la décision de sauvegarder son état. Le principal avantage de la sauvegarde non coordonnée est qu'un processus peut procéder à la sauvegarde de son état lorsque celui-ci est minimal, réduisant ainsi le sûrcout de la sauvegarde en terme de quantité d'informations à sauvegarder [124].

Mais l'inconvénient principal de cette approche est le risque d'effet domino [100] qui peut causer une perte importante du travail réalisé avant la panne et la sauvegarde inutile de

points de reprise, ceux-ci ne faisant jamais partie d'un état global cohérent. Ces points de reprise augmentent le coût de l'exécution normale (i.e. en l'absence de panne) et ne servent pas à la reprise après panne. De plus, la sauvegarde non coordonnée oblige chaque processus à maintenir plusieurs points de reprise.

Afin de déterminer un état global cohérent (ligne de recouvrement), chaque processus dispose d'un journal en mémoire stable dans lequel il enregistre tout ou partie des messages échangés ainsi que leurs historiques [13]. Lorsqu'une défaillance survient, l'algorithme de reprise utilise les points de reprise locaux et les journaux afin de déterminer une ligne de recouvrement.

3.7.3 Sauvegarde induite par les communications

La sauvegarde induite par les communications (*Communication-Induced Checkpointing CIC*) [7], est un compromis entre la sauvegarde coordonnée et la sauvegarde non coordonnée.

L'idée principale est d'une part, d'éviter la coordination des processus en permettant la sauvegarde non coordonnée des processus, appelée **sauvegarde locale**, et d'autre part, d'éviter l'effet domino en forçant un processus à sauvegarder son état en cas d'évaluation de la ligne de recouvrement. Cette sauvegarde est alors appelée **sauvegarde forcée**.

Pour résumer, le principe de CIC est d'étendre un ensemble de points de reprise locale (sauvegardes locales) de l'application (système) à un ensemble de points de reprise constituant un état global cohérent (sauvegarde globale). Ce principe a été proposé formellement dans [92], par la définition des chemins zigzagants qui formalise les conditions de cohérence d'un état global.

Un Z-chemin (**zigzag path**) est une séquence spéciale de messages qui connecte deux points de reprise [92]. On note $Envoie_i(m_l)$ (resp. $Reception_i(m_l)$) l'évènement d'envoi (resp. réception) du message m_l par le processus p_i. Soit $c_{i,x}$ le x^{ieme} point de reprise sur le processus P_i. Supposons que la partie d'exécution entre deux points de reprise consécutifs sur le même processus est l'intervalle de sauvegarde. Étant donnés deux points de reprise $c_{i,x}$ et $c_{j,y}$, il existe un **Z-chemin** entre $c_{i,x}$ et $c_{j,y}$ ssi l'une des deux conditions suivantes est vraie :

1. $x < y$ et $i = j$;
2. il existe une séquence de messages $[m_0, m_1, ..., m_n]$ tel que :
   - $c_{i,x}$ précède causalement $Envoie_i(m_0)$.
   - $\forall l < n$, il existe k tel que : soit $Reception_k(m_l)$ et $Envoie_k(m_{l+1})$ sont dans le même intervalle de sauvegarde ; soit,
     $Reception_k(m_l)$ précède causalement $Envoie_k(m_{l+1})$.
   - $Reception_j(m_n)$ précède causalement $c_{j,y}$.

Par définition, un **Z-cycle** est un Z-chemin qui commence et termine avec le même point de reprise [92]. Dans le contexte du protocole CIC, les notions de Z-chemin et de Z-cycle sont très intéressantes car on peut prouver qu'un point de reprise est inutile si et seulement si il fait partie d'un Z-cycle [92]. En conséquence, une façon d'éviter les points de reprise inutiles est d'assurer qu'un Z-chemin ne devient jamais un Z-cycle.

3.8 Critères de comparaison

Comme nous l'avons vu dans la section précédente, il existe plusieurs protocoles de tolérance aux pannes basés sur une mémoire stable. Tous ces protocoles se basent sur la sauvegarde d'un état des processus du système d'une part, et sur la construction d'un état global cohérent d'autre part. Sept critères fondamentaux permettent alors de comparer ces différents protocoles :

1. **état sauvegardé** : suivant que l'état sauvegardé consiste en des données en mémoire et du code (*état faible*), ou s'il contient aussi l'état d'exécution des processus en cours (*état fort*) ;
2. **coordination** : si les processus se coordonnent afin de construire un état global cohérent au moment de la sauvegarde ou pas ;
3. **"multithreadé"** : si le mécanisme de sauvegarde permet de sauvegarder et de restaurer les applications "multithreadées" ;
4. **hétérogénéité** : si l'état sauvegardé peut être restauré sur un ensemble varié de processeurs et de systèmes d'exploitation, le protocole est dit *hétérogène*. Dans le cas contraire, il est dit *homogène* ;
5. **reprise globale ou locale** : la restauration après une défaillance nécessite-t-elle la construction d'un état cohérent global pour redémarrer l'application ? Si oui, le protocole est dit à *restauration globale* ; si non le protocole est dit à *restauration locale*, et seule une connaissance de l'état du processus avant défaillance, voire des processus dans son voisinage local, suffit à permettre le redémarrage de l'application ;
6. **remplacement d'une ressource défaillante** : faut-il remplacer une ressource défaillante par une nouvelle ressource ou bien peut-on restaurer l'état de la ressource défaillante sur une autre qui existe déjà ?
7. **composants fiables durant toute l'exécution** : De combien de composants fiables le système tolérant aux pannes doit disposer pour pouvoir traiter les pannes considérées ?

Ces critères permettent de donner une taxonomie des différents protocoles pour la sauvegarde / reprise. Néanmoins ces critères doivent être complétés afin de permettre leur comparaison lors d'une utilisation sur une grille de calcul. La performance de ces protocoles est fondamentale pour une exploitation efficace des architectures de type grille. Nous nous intéresserons à deux métriques. La première concerne le **surcoût** en temps et en mémoire nécessaire à l'exécution d'une application avec un protocole basé sur mémoire stable. La seconde concerne le **passage à l'échelle** de ces protocoles sur plusieurs (centaines de) milliers de processeurs.

3.9 Tolérance aux pannes par reprise dans les systèmes existants

Plusieurs systèmes ont été rendus tolérants aux pannes par sauvegarde/reprise. Dans cette section, nous présentons quelques systèmes tolérants aux pannes dans le domaine du parallèlisme, selon les critères de comparaisons précédents (§3.8).

3.9.1 Condor

Condor [83] est un système de traitement par lots conçu pour être utilisé sur des systèmes UNIX ; il permet la répartition des tâches sur un ensemble de ressources de calcul.

Condor supporte un mécanisme de sauvegarde transparent [84] basé sur la sauvegarde de l'espace d'adressage d'un processus. Les points de reprise réalisés par Condor peuvent alors servir à la migration de processus et à la reprise après panne. Mais migration et reprise doivent être effectuées sur des ressources homogènes puisqu'un point de reprise est un espace d'adressage du processus.

Dans Condor, toutes les fonctions de sauvegarde sont réalisées au niveau utilisateur. Afin d'utiliser la sauvegarde de Condor dans une application, l'édition de liens doit être refaite afin d'inclure la bibliothèque Condor. Condor ne supporte pas les applications utilisant des communications, ni les applications multithreadées.

Comme l'état sauvegardé est un espace d'adressage de processus, l'opération de reprise nécessite une nouvelle ressource (processus) pour remplacer un processus défaillant.

3.9.2 CoCheck

Le système CoCheck [113], propose une extension de la sauvegarde de Condor afin d'ajouter le mécanisme de sauvegarde pour des applications parallèles.

CoCheck utilise la bibliothèque Condor pour sauvegarder l'état d'un processus. La cohérence d'état global de l'application parallèle est assurée par l'utilisation de techniques de sauvegarde coordonnée basées sur l'algorithme de Chandy et Lamport [34]. En conséquence, CoCheck suppose la reprise globale de l'application.

Comme Condor, CoCheck ne supporte pas les applications "multithreadées" et nécessite une nouvelle ressource homogène avec celle du processus défaillant pour effectuer sa reprise.

3.9.3 Netsolve

Netsolve [95] est un environnement logiciel dont l'objectif est de faciliter l'utilisation des systèmes de grille. Dans cette optique, Netsolve supporte des mécanismes de tolérance aux pannes et de répartition de charge transparents basés sur la sauvegarde/reprise.

Netsolve offre en coopération avec Condor des services de migration de processus ainsi que la tolérance aux pannes d'applications parallèles par une sauvegarde coordonnée. Ainsi, le mécanisme de sauvegarde/reprise adopté ne supporte pas les applications "multithreadées" et nécessite encore une fois une nouvelle ressource homogène avec celle du processus défaillant pour pouvoir effectuer la reprise de ce dernier.

3.9.4 Porch

Porch [115] est un compilateur source à source qui traduit des programmes séquentiels C en des programmes sémantiquement équivalents qui sont capables d'effectuer la sauvegarde de points de reprise portables. Ces points de reprise permettent la reprise après panne sur des machines incompatibles (hétérogènes) grâce à l'état sauvegardé dans un point de reprise qui contient la pile des appels de procédure et les valeurs des variables (globales). La sauvegarde d'un point de reprise est faite sur la réception d'un signal de sauvegarde. Porch ne supporte pas les applications parallèles et donc aucune notion de cohérence d'état global n'est définie.

3.9.5 MPICH-V

MPICH-V1 [19] est basé sur la sauvegarde non coordonnée avec journalisation des messages pour des programmes parallèles écrits avec MPI. Afin de pouvoir redémarrer uniquement les processus défaillants, MPICH-V1 utilise le concept de canal mémoire ; tous les messages échangés passent par un canal mémoire supposé fiable. Cette technique implique qu'il n'est pas possible d'avoir de communications directes entre les processus.

Pour éviter l'utilisation du canal mémoire, MPICH-causal [20] implante une journalisation causale des événements non déterministes (messages).

MPICH-cl [22] supporte un protocole de sauvegarde coordonnée. Afin d'améliorer la performance d'une reprise globale [21], chaque processus stocke ses points de reprise en local et sur mémoire stable.

Les trois versions de MPICH-V utilisent la bibliothèque de points de reprise fournie par Condor [84] et donc ne supportent ni l'hétérogénéité ni le "multithreadé".

3.9.6 Cilk

Cilk [15] est un langage de programmation parallèle basé sur le langage C et destiné aux machines à mémoire partagée. Le parallélisme et les synchronisations sont décrites explicitement par l'utilisateur. Le modèle de programmation de Cilk est de type série parallèle (*i.e fork/join*). La stratégie d'ordonnancement est de type vol de travail [14, 49] ; lorsqu'un processeur devient inactif, il demande du travail à un autre processeur.

Cilk-NOW [17] est une tentative d'implantation du langage Cilk de manière distribuée et tolérante aux pannes (franches). Le principe de base adopté pour la tolérance aux pannes est de refaire le travail du processus défaillant. Pour limiter la quantité de travail à refaire suite à une panne, chaque processus réalise automatiquement de points de reprise sans tenir compte de la cohérence globale d'état de l'application. Afin de garantir une exécution cohérente en présence de pannes, l'exécution d'une tâche volée par un processus est vue comme une transaction. L'environnement Atlas [6], permettant l'implantation des applications parallèles de type "diviser pour régner" et son mécanisme de tolérance aux pannes sont basés sur Cilk-NOW.

3.9.7 Charm++

Charm++ [67] est un langage de programmation parallèle par objet basé sur le langage C++. La description du parallélisme dans Charm++ est basée sur la création d'objets concurrents appelés **chares**. Un objet **chare** est un objet C++ standard dont les méthodes peuvent être appelées à distance par un autre **chare** disposant d'une référence sur cet objet.

Charm++ supporte un mécanisme de tolérance aux pannes franches [129, 32] par sauvegarde coordonnée des états de tous les processus participant au calcul. L'état du processus sauvegardé est l'ensemble des **chares** qu'il contient.

Après une panne, la reprise globale de tous les processus est imposée par la sauvegarde coordonnée. Afin de réduire le surcoût d'une reprise, les points de reprise de chaque processus sont stockés en local et une copie de ces points est stockée à distance sur un autre processus de calcul qui est considéré comme un serveur de stockage pour le premier processus. Dans le cas où un processus est en panne en même temps que son serveur de stockage, le protocole proposé ne fonctionne plus.

3.9.8 ProActive

ProActive [9] est une bibliothèque Java pour le calcul distribué basée sur la notion d'*objets actifs*. ProActive propose un protocole de tolérance aux pannes par sauvegarde induite par les communications (CIC). L'état sauvegardé est constitué de chacun des objets actifs dans des états bien définis. ProActive implante une *reprise globale* : la restauration après panne nécessite l'arrêt et la reprise de tous les processus.

3.9.9 Satin

Satin [123] est un environnement de programmation parallèle basé sur la méthode récursive “diviser pour régner". Dans Satin, chaque processeur maintient une file de travail qui contient les tâches à exécuter. L'algorithme d'ordonnancement utilisé est un algorithme dynamique basé sur le vol de travail (*work stealing*). Ce système a été implanté en Java.

Satin propose un mécanisme de tolérance aux pannes [128, 127] pour ses applications. Comme decrit dans [47, 56, 80], ce mécanisme exploite le modèle de programmation (série parallèle) qui peut être vu comme une généralisation du modèle maître-esclave traditionnel. L'approche est basée sur l'utilisation d'une table globale pour stocker les résultats potentiels des tâches orphelines : une tâche est dite orpheline si elle a été volée par un processeur qui est tombé en panne par la suite. L'intérêt d'une telle solution est d'éviter la perte et la redondance du travail en cas de panne. L'approche choisie nécessite d'une part, l'identification de toutes les tâches du programme de manière globale, unique et reproductible, et suppose une exécution déterministe d'autre part (i.e. deux exécutions parallèles génèrent et exécutent les mêmes tâches). Satin ne traite pas le cas d'une panne globale sur tous les processus car la panne de tous les processus a pour conséquence la perte de la table globale et donc le redémarrage de l'application à partir de l'état initial.

Pour identifier les tâches orphelines suite à une panne, chaque processeur maintient une liste des tâches qui lui ont été volées ainsi que l'auteur de chaque vol. Une fois une panne détectée, chaque processeur vivant exécute une procédure de recouvrement qui consiste à annuler les opérations de vol effectuées par le processeur défaillant et à marquer toutes les tâches concernées, par l'annulation de vols, comme étant "à ré-exécuter". Quand un processeur veut exécuter une tâche marquée "à re-exécuter", il consulte préalablement d'abord la table globale des résultats afin de vérifier que le résultat de cette tâche n'est pas déjà stocké dans la table globale.

Satin adopte un modèle proche de notre modèle de programmation est utilisé pour comparer notre protocole de tolérance aux pannes au chapitre 6.

3.10 Bilan & Conclusion

Dans ce chapitre, nous avons présenté les techniques de tolérance aux pannes basées sur une mémoire stable dans un système réparti / parallèle (voir figure 3.3). Nous avons vu dans un premier temps que le recouvrement d'erreur par reprise pose deux problèmes : le déterminisme d'exécution et la cohérence de l'état global.

Dans le cas où l'exécution respecte l'hypothèse PWD, une des stratégies de journalisation des événements non déterministes peut être utilisée avec ou sans sauvegarde périodique de l'état des processus participants. Si l'exécution ne respecte pas cet hypothèse, une autre

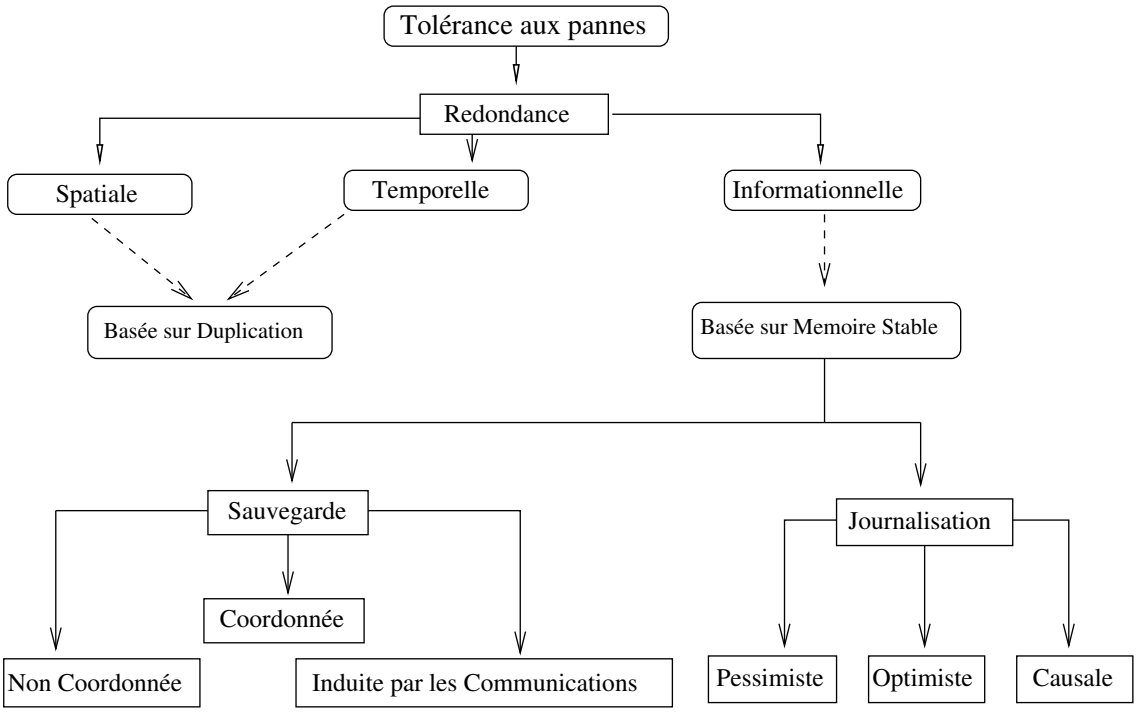

Figure 3.3 – Tolérance aux pannes par mémoire stable

stratégie de sauvegarde, présentée dans §3.7, doit être utilisée. La cohérence d'état global peut être assurée soit au moment de la sauvegarde des points de reprise, soit durant l'opération de reprise. Le protocole de sauvegarde ou de journalisation détermine si après une panne, la reprise globale du système est imposée ou si seuls les processus defaillants doivent effectuer une opération de reprise.

Le contenu d'un point de reprise ou l'état sauvegardé par un processus joue un rôle très important dans un système tolérant les pannes car ce contenu peut déterminer, d'une part, si la reprise d'un processus nécessite d'allouer une nouvelle ressource ou si elle peut être faite sur une ressource déja allouée. D'autre part ce contenu détermine si le processus défaillant peut ou non être remplacé par un autre sur une machine compatible à celle d'où provient le processus défaillant. Le tableau 3.1 récapitule les avantages et les inconvénients des différentes techniques de tolérance aux pannes basées sur une mémoire stable.

Dans l'étude des différents systèmes tolérant les pannes franches se basant sur le concept de mémoire stable, nous avons vu que les outils de sauvegarde proposés ont été réalisés pour des utilisations spécifiques dont l'évaluation reste seulement expérimentale. Ils ne sont généralement pas réutilisables. De plus, pour la plupart, l'état sauvegardé ne permet pas de supporter l'hétérogénéité [84, 19, 20]. Deux systèmes pallient ce défaut : ProActive [9], car le langage Java fournit un environnement d'exécution portable et Porch [115], parce que l'outil s'insère dans le processus de compilation de l'application. Ajoutons que la plupart de ces systèmes supposent que l'exécution se déroule sur un nombre fixé de processus et nécessitent un nouveau processus en remplacement du processus défaillant

Le tableau 3.2 synthétise la comparaison des systèmes décrits dans §3.9 selon les critères présentés dans §3.8.

L'objectif de notre travail de thèse est l'exploitation fiable et efficace d'une plateforme

dynamique et hétérogène (éventuellement SMP). Ceci nécessite un mécanisme de tolérance aux pannes performant supportant l'ajout et la suppression de ressources hétérogènes. Ce travail est réalisé au niveau applicatif au sein de l'environnement KAAPI présenté dans le chapitre suivant.

	Journalisation Pessimiste	Journalisation Optimiste	Journalisation Causale	Sauvegarde Coordonnée	Sauvegarde Non Coordonnée	Sauvegarde Induite par les Com.
Hypothèse PWD	Oui	Oui	Oui	Non	Non	Non
Effet Domino	Non	Non	Non	Non	Possible	Non
Point de reprise par processus	Un	Plusieurs	Un	Un	Plusieurs	Plusieurs
Processus Orphelin	Non	Possible	Non	Non	Possible	Possible
Propagation de Reprise	dernier point de reprise	plusieurs points de reprise ?	dernier point de reprise	dernier point de reprise	non bornée	plusieurs points de reprise ?

Tableau 3.1 – Comparaison entre les différents techniques de la reprise

	État sauvegardé	Coordination	Multithreadé	Hétérogénéité	Restauration globale ou locale	Remplacement de ressource défaillante	Composants fiables	Applications
Condor	Image mémoire	-	Non	Non	-	nouvelle ressource	mémoire stable	Séquentielle
CoCheck	Image mémoire	Oui	Non	Non	Globale	nouvelle ressource	mémoire stable	Parallèle
Netsolve	Image mémoire	Oui	Non	Non	Globale	nouvelle ressource	mémoire stable	Parallèle
Porch	Appels de procédures + variables globales	-	-	Oui	-	-	mémoire stable	Séquentielle
Mpich-v1	Image mémoire	Non	Non	Non	Locale	nouvelle ressource	mémoire stable + Canal memoire + Dispatcher	Parallèle (MPI)
Mpich-cl	Image mémoire	Oui	Non	Non	Globale	nouvelle ressource	mémoire stable + Dispatcher	Parallèle (MPI)
Mpich-causal	Image mémoire	Non	Non	Non	Locale	nouvelle ressource	mémoire stable + Dispatcher	Parallèle (MPI)
Cilk	Tâches (jobs)	Non	Oui	Oui	Locale	-	mémoire stable	Parallèle série parallèle
Charm++	Chares	Oui	Oui	Non	Globale	-	mémoire stable	Parallèle
ProActive	Objets Actifs	Non	Oui	Oui	Globale	nouvelle ressource	mémoire stable	Parallèle
Satin	Tâches (jobs)	Non	Oui	Oui	Locale	-	table globale	Parallèle série parallèle

Tableau 3.2 – Tolérance aux pannes par reprise dans les systèmes existants : synthèse.

Deuxième partie

Tolérance aux pannes basée sur la sauvegarde/reprise d'un graphe de flot de données

Chapitre 4

Modèle de programmation & d'exécution

Sommaire

4.1 Introduction

Dans ce chapitre, nous décrivons le modèle de programmation et d'exécution (*system model*) qui est la cible de nos protocoles de tolérance aux pannes.

Afin de faciliter la programmation parallèle en augmentant la portabilité des applications, plusieurs modèles de programmation ont été proposés qui sont plus abstraits que les modèles de programmation bas niveau des architectures parallèles, tels PVM [54] ou MPI [57], basés sur le multithreading pour exploiter le parallélisme intra-nœud multiprocesseur et l'échange de messages inter-nœuds [29]. Ces modèles sont basés sur l'exploitation du parallélisme intrinsèque, qu'il soit de contrôle (i.e. parallélisme fonctionnel comme dans Cilk [15] ou Satin [123]) ou de données (i.e. parallélisme de tableaux comme dans HPF [48] et BSP [122] ou plus généralement, les langages "flot de données" comme Mentat [93], Jade [102] ou ATHAPASCAN /KAAPI [49]). Ces modèles sont communément désignés par "modèles de programmation parallèle haut niveau" : le programme applicatif explicite un grand degré de parallélisme potentiel indépendamment de l'architecture ; ce parallélisme potentiel est replié par le système d'exécution en exploitant un parallélisme effectif adapté à l'architecture. L'intérêt de ces modèles est de faciliter la programmation parallèle : le programmeur explicite les tâches et les dépendances de données associées, qui sont indépendantes de l'architecture ; mais il ne gère pas explicitement les synchronisations qui doivent être mises en œuvre lors d'une exécution pour garantir le respect des dépendances entre tâches.

Le modèle de programmation parallèle visé par ce travail est un modèle haut niveau où la détection du parallélisme de l'application est réalisée dynamiquement en cours d'exécution par analyse des dépendances de données du programme. Cette analyse est réalisée à un niveau macroscopique, celui des tâches : une tâche correspond à une séquence d'instructions (appel de procédure). L'exécution du programme est alors explicitement décomposée en tâches de calcul par le programmeur. L'extraction du parallélisme entre ces tâches nécessite d'analyser les dépendances de données entre les tâches. Grâce à cette analyse de dépendances de données, le système peut maintenir pendant l'exécution une représentation abstraite de l'état de l'exécution, typiquement sous la forme d'un graphe décrivant les tâches et leurs dépendances de données : on parle dans la suite de *graphe de flot de données*.

Le système d'exécution repose sur une machine virtuelle (i.e. des mécanismes d'abstraction de la machine parallèle) pour exécuter ces tâches en respectant les contraintes de précédence induites par les dépendances de données. La description du parallélisme est alors indépendante de l'architecture distribuée sous-jacente : un algorithme d'ordonnancement décide de la machine cible chargée de l'exécution des tâches et du stockage des données. La figure 4.1 présente le schéma général du modèle de programmation parallèle cible de ce travail de thèse.

Ainsi, le modèle de programmation parallèle haut niveau est basé sur une représentation abstraite de l'état d'exécution sous forme d'un graphe de flot de données. Cette représentation constitue un état global de l'application ; ainsi, la sauvegarde de cette représentation abstraite sur mémoire stable permet la réalisation de la tolérance aux pannes pour ce modèle.

Le reste de ce chapitre présente d'abord le modèle de programmation parallèle cible. Puis, le principe de l'ordonnancement par vol de travail et un modèle de coût associé sont présentés. Enfin, nous présentons l'intergiciel KAAPI , une instance du modèle cible, qui constitue le cadre expérimental de ce travail.

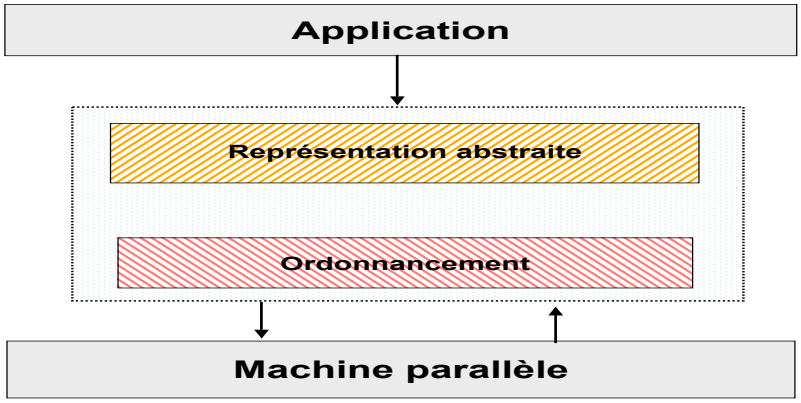

Figure 4.1 – Schéma général du modèle de programmation parallèle cible.

4.2 Modèle cible et représentation abstraite par graphe de flot de données

Dans le modèle de programmation parallèle considéré, la détection du parallélisme de l'application est réalisée dynamiquement en cours d'exécution par analyse des dépendances de données du programme. Cette analyse de dépendances de données permet de construire pendant l'exécution une représentation abstraite de l'application sous forme d'un graphe.

La sémantique et la forme de ce graphe dépend du langage haut-niveau considéré et des primitives qu'il propose pour exprimer le parallélisme et les synchronisations.

- dans Jade [102, 103] et ATHAPASCAN [49, 43, 50, 30, 101] des mécanismes permettent d'indiquer au système d'exécution, lors de la création d'une tâche, l'ensemble des effets de bords pouvant être réalisés par celle-ci, et donc l'ensemble des dépendances de données avec les autres tâches. Dans ces langages, le graphe qui constitue la représentation abstraite de l'exécution décrit les tâches, les données (en assignation unique) et les dépendances de lecture-écriture entre tâches et données ; on parle de *graphe de flot de données*.
- dans Cilk [15] et Satin [123], les primitives `spawn` et `sync` permettent respectivement de décrire une tâche parallèle et de se synchroniser sur les tâches précédemment créées. La représentation abstraite est alors restreinte à un graphe de tâches type série-parallèle (*i.e. fork/join*) représentant les précédences entre les tâches non encore terminées. Une tâche ne peut être démarrée que lorsque toutes les tâches qui la précèdent son terminées. Pour garantir la sémantique des accès en lecture et écriture aux données accédées, Cilk repose sur une mémoire implémentant une cohérence de type séquentiel (*DAG consistency*) ; Satin restreint quant à lui les tâches à ne prendre que des paramètres par valeur et renvoyer un unique résultat, sans effet de bord.

Dans la suite, nous considérons le cas le plus général où la représentation abstraite de l'exécution est un graphe de flot de données. Ce graphe est défini dans la section suivante.

4.2.1 Graphe de flot de données

Le graphe de flot de données associé à une exécution est défini comme suit.

Définition 3 *Le **graphe de flot de données** associé à une exécution est le graphe $G = (\mathcal{V}, \mathcal{E})$, tel que les tâches $\mathcal{V}_t$ et les données $\mathcal{V}_d$ forment l'ensemble des nœuds du graphe $\mathcal{V} = \mathcal{V}_t \cup \mathcal{V}_d$, et que les accès des tâches aux données forment l'ensemble $\mathcal{E}$ des arêtes du graphe. A chaque nœud tâche (respectivement donnée) est associé une séquence d'instructions – corps de procédure – (respectivement une adresse logique) lui correspondant. Ce graphe est orienté et biparti : un nœud tâche est connecté à un ou plusieurs nœuds données ; un nœud donnée est connecté à une ou plusieurs tâches. Une arête dans le graphe signifie une synchronisation de type lecture ou écriture entre une donnée et une tâche ; soient $t \in \mathcal{V}_t$ une tâche et $d \in \mathcal{V}_d$ une donnée, alors :*

- *l'arête $(t, d) \in \mathcal{E}$ signifie un droit d'accès en écriture de la tâche t sur la donnée d ; la tâche t précède toute tâche t' qui accède d en lecture.*
- *l'arête $(d, t) \in \mathcal{E}$ un droit d'accès en lecture de la tâche t sur la donnée d ; la tâche t est précédée par toute tâche t' qui accède d en écriture.*

De manière générale, les écritures peuvent être concurrentes : plusieurs tâches peuvent accéder une même donnée en écriture. Dans ce cas, la valeur produite (qui sera lue par toutes les tâches successeurs) est la fermeture transitive par une opération associative d'accumulation des valeurs élémentaires écrites.

Un exemple de graphe de flot de données représentant les accès de 3 tâches à 5 données est présenté dans la figure 4.2 : la tâche t_1 accède d'abord aux données d_1 et d_2 en lecture et ensuite aux données d_3 et d_4 en écriture, les tâches t_2 et t_3 accèdent aux données d_3 et d_4 en lecture et ensuite à la donnée d_5 en écriture accumulée.

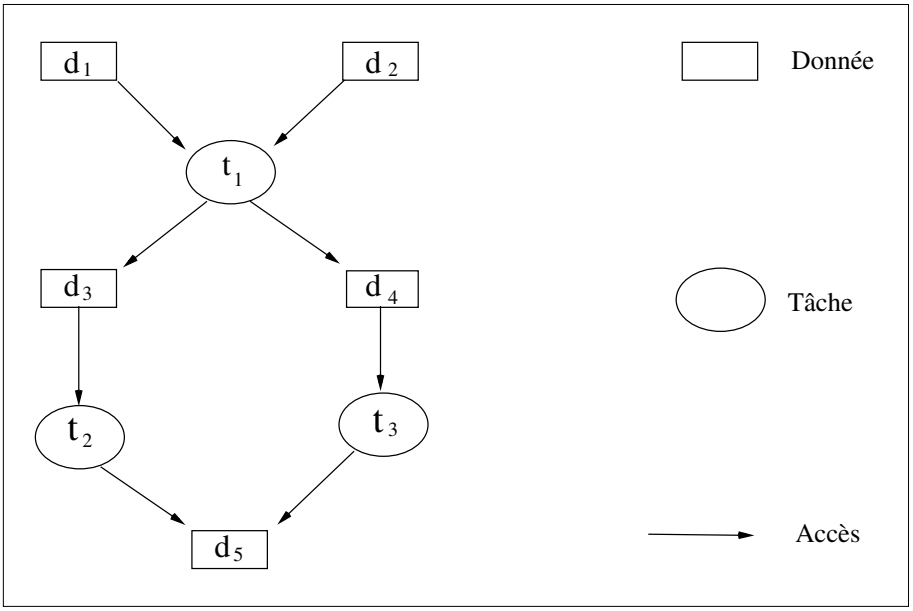

Figure 4.2 – Graphe de flot de données.

4.2.2 Modèle général de programmation et d'exécution

Dans le modèle de programmation considéré, le parallélisme d'une application est décrit explicitement par le programmeur :

Au niveau du modèle de programmation, nous supposons que le programme dispose de primitives permettant de créer des tâches avec dépendances de données : lors de la création d'une tâche t, l'ensemble des données que t accède d'une part en lecture et d'autre part en écriture, sont explicitées par le programme. Au niveau du modèle d'exécution, afin d'exploiter le parallélisme, une analyse des dépendances de données entre les tâches est réalisée. Cette analyse construit une représentation abstraite, sous forme d'un graphe de flot de données. Ce graphe est ensuite utilisé pour décider de l'ordre d'exécution et du placement des tâches de calcul sur la machine parallèle (i.e. en appliquant des algorithmes d'ordonnancement).

Par conséquent, l'exécution des programmes parallèles, est décomposée en deux étapes : la première sert à construire une représentation abstraite sous forme de graphe de flot de données de l'état courant de l'exécution du programme, c'est à dire les tâches en cours d'exécution et celles restant à exécuter avec leurs dépendances de données ; la seconde utilise cette représentation abstraite pour décider de l'ordre d'exécution et du placement des tâches de calcul (ordonnancement).

Dans les environnements de programmation parallèle, adoptant cette approche de programmation, on peut distinguer deux niveaux pour la réalisation des deux étapes précédentes (figure 4.3) :

- le premier est le niveau «applicatif» (API) du modèle et est noté dans la suite **niveau 0**. Il s'agit de l'interprétation du programme dans le but soit de produire des sorties (les résultats du calcul), soit de construire la représentation abstraite. Une machine virtuelle (MV 0) interprète les instructions d'un langage appelé *langage API ou L0* (défini par l'environnement de programmation parallèle) qui permet la description du parallélisme de l'application par la création des tâches avec leurs dépendances des données.
- le second est le niveau «ordonnancement» qui est noté dans la suite **niveau 1**. Ce niveau utilise la représentation abstraite fournie par le niveau 0 (API) pour réaliser une exécution sur une architecture parallèle en décidant de l'ordonnancement des tâches du programme. Une machine virtuelle (MV 1) interprète les instructions d'un langage appelé *langage L1* (défini par l'environnement de programmation parallèle). Ce langage, L1, permet de décrire l'algorithme d'ordonnancement.

L'exécution proprement dite se fait au niveau «applicatif» (niveau 0) lors de l'exécution du corps des tâches du programme. La figure 4.3 montre d'une manière générale les deux niveaux d'exécution d'un programme dans le modèle de programmation et d'exécution cible de cette thèse.

4.3 Représentation abstraite et tolérance aux pannes

Afin de réaliser la tolérance aux pannes d'une application sur le modèle précédent, il est nécessaire de définir un état global permettant le calcul de points de reprise.

Le modèle de programmation et d'exécution considéré repose sur une représentation

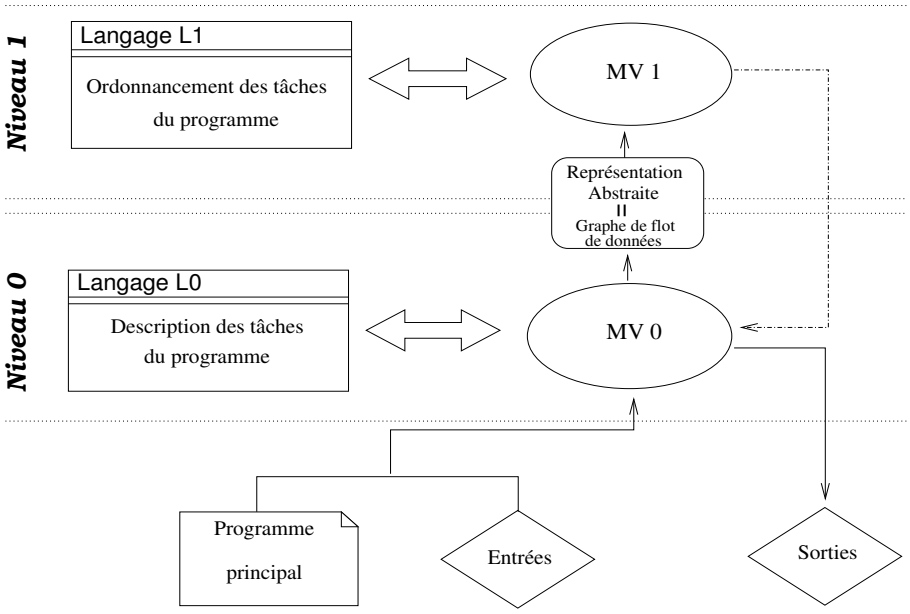

Figure 4.3 – Schéma général d'exécution des programmes parallèles.

abstraite de l'application afin de permettre une exécution parallèle indépendamment de la machine parallèle. Puisque cette représentation décrit à tout instant l'ensemble des tâches en cours et restant à ordonnancer, une question d'importance est la suivante : « peut-on utiliser cette représentation pour définir un état global d'une exécution distribuée ?» Une réponse positive permettra de l'utiliser pour réaliser la sauvegarde/reprise d'une exécution distribuée.

Les conditions nécessaires et suffisantes qui caractérisent une représentation d'un état global d'une exécution distribuée sont :

1. la représentation doit être **close** : c'est-à-dire qu'elle ne dépend pas de données externes. Autrement dit, l'état d'un processus peut se ramener à l'état de cette représentation à certaines dates ;
2. la représentation doit être **causalement connectée** à l'application. En d'autre terme il faut que cette représentation soit un objet représentant fidèlement l'exécution de l'application : toute modification de l'état de l'exécution correspond à une modification de la représentation et, inversement, toute modification de la représentation doit être répercutée sur l'exécution de l'application.

La représentation abstraite sous forme de graphe de flot de données définie précédemment vérifie ces deux propriétés :

- elle est close, puisqu'à tout instant, pour chaque tâche créée qui est en cours ou non encore exécutée, le graphe décrit toutes ses dépendances de données et la séquence d'instructions qui lui est associée. Sous la réserve de la définition de formats portables des noeuds du graphe, cette représentation est indépendante de l'architecture et supporte l'hétérogénéité.
- elle est causalement connectée, puisque, par définition, elle est l'unique interface entre l'application (niveau 0) et l'ordonnancement (niveau 1).

Donc, la représentation précédente par graphe de flot de données peut être utilisée pour construire un état global d'une exécution distribuée. Dans la suite, nous utiliserons cette re-

présentation de l'état global par graphe de flot de données comme représentation de base pour réaliser la sauvegarde/reprise. Cette représentation est évidemment insuffisante puisqu'elle ne garantit pas la cohérence de l'état global associé à la représentation. Nous verrons dans le chapitre suivant des protocoles permettant de garantir un état global cohérent.

Ainsi, la représentation par graphe de flot de données fournit un état global ; afin de l'utiliser pour réaliser des points de reprise, il est nécessaire de capturer cette représentation et de la stocker sur mémoire stable. Par rapport aux deux niveaux L0/API et L1/Ordonnacement décrits précédemment, deux méthodes extrêmes peuvent être appliquées afin de faire cette capture :

- la première consiste à capturer la représentation au niveau 0/API lors de la construction du graphe de flot de données. Cette méthode se base sur la modification de l'interprétation du langage L0 pour ajouter la capture de l'état.
- la deuxième consiste à capturer la représentation au niveau 1/Ordonnacement lors de l'évaluation du graphe de flot de données. Cette méthode se base sur la modification de l'interprétation du langage L1 pour ajouter la capture de l'état.

Cette deuxième méthode est particulièrement liée à l'ordonnancement effectué au niveau 1 ; son intérêt est de placer le surcoût de capture au niveau des opérations de modification de l'ordonnancement réalisées au niveau 1. Or, certains ordonnancements n'effectuent qu'un nombre limité de telles opérations ; c'est le cas des ordonnancements par vol de travail décrits dans la section suivante.

4.4 Ordonnancement par vol de travail et modèle de coût

Puisque la représentation abstraite de l'exécution des programmes parallèles, dans le modèle considéré, est construite dynamiquement au cours de l'exécution, les algorithmes d'ordonnancement utilisés sont donc nécessairement dynamiques[1].

Parmi les algorithmes d'ordonnancement dynamique, les ordonnancements par vol de travail (*workstealing*) sont très utilisés par les environnements de programmation de haut niveau [14, 49]. Nous décrivons dans cette section le principe d'un tel ordonnacement ainsi que les résultats théoriques permettant d'en analyser le coût et la performance.

De manière générale, un algorithme par vol de travail suit le principe glouton (*greedy schedule*) : à tout instant où il existe une tâche prête mais non encore ordonnancée, tous les processeurs sont actifs. Parmi les ordonnancements gloutons figurent les ordonnancements dits de liste : les tâches prêtes non encore ordonnancées sont stockées dans une liste ; lorsqu'un processeur devient inactif, il va chercher une tâche prête dans cette liste si celle-ci est non vide et sinon il s'ajoute à la liste des processeurs inactifs.

Les ordonnancements par vol de travail suivent ce principe mais sont de plus distribués : chaque processeur gère localement la liste des tâches que lui-même a créées dans une pile. Chaque fois qu'un processeur crée une tâche, il l'empile dans sa pile en dernière position. Lorsqu'un processeur termine une tâche, deux cas sont distingués :

- soit sa pile locale contient des tâches prêtes : dans ce cas il prend la plus récente parmi celles-ci (ordre LIFO) et démarre localement son exécution ;

[1] Cependant dans certains cas, des algorithmes d'ordonnancement statiques peuvent être utilisés. Par exemple, dans le cas où un programme crée de manière itérative l'ensemble des tâches du programme.

– soit sa pile locale est vide ou ne contient pas de tâches prêtes, auquel cas il passe à l'état «voleur» et cherche à récupérer du travail sur les autres processus. Cet état reste inchangé tant qu'il n'a pas trouvé un autre processeur, appelé «victime» possédant au moins une tâche prête. Dans ce cas, il vole au processeur victime sa tâche prête la plus ancienne (ordre FIFO au niveau du vol).

Le choix du processeur victime peut être fait de différentes manières. Dans la suite, nous considérons que ce choix est fait de manière aléatoire (*random workstealing*). L'intérêt de ce choix est qu'il peut être implanté efficacement en distribué sans nécessiter de synchronisation, tout en garantissant une performance quasi optimale avec une très grande probabilité[2]. L'environnement KAAPI implémente en particulier un tel ordonnancement.

La suite de cette section résume les notations et résultats principaux concernant la performance théorique d'un ordonnancement par vol de travail, distribué et aléatoire. Cette performance est exprimée à partir de mesures théoriques de coût associées à une application parallèle écrite dans un langage de haut-niveau, de type L0/API décrit précédemment. Ces mesures sont définies ci-dessous.

Définition 4 (Mesures théoriques de coût : T_s, T_1 et T_∞) *A l'exécution théorique d'un programme dans le langage de haut-niveau L0/API sur une machine parallèle avec une infinité de processeurs identiques sont définies les coûts algorithmiques suivants, qui sont associés au graphe de flot de données total (et sa construction) :*

– *T_s est la durée d'exécution de l'algorithme séquentiel sur un seul processeur. T_s traduit le coût de la méthode de calcul utilisée par l'application et ne contient aucun surcoût lié au parallélisme ;*
– *T_1 est la durée d'exécution de l'algorithme parallèle sur un seul processeur. T_1 représente le travail de l'application et contient, outre le coût T_s de la méthode du calcul utilisée, le surcoût de description du parallélisme (opérations du langage L0 qui l'implemente).*
– *T_∞ est la durée d'exécution théorique sur une infinité de processeurs. Elle exprime le maximum des durées d'exécution des tâches qui doivent être exécutées séquentiellement afin de respecter les contraintes de précédence. T_∞ représente alors la durée d'exécution d'un plus long chemin dans le graphe de flot de données et ne contient pas le surcoût d'ordonnancement.*

Pour analyser la performance de l'algorithme par vol de travail sur une machine composée de p processeurs identiques, on considère trois mesures :

– T_p, la durée d'exécution de l'algorithme parallèle sur p processeurs ;
– σ, la taille du graphe de flot de données G représentant l'exécution totale de l'application, c'est à dire le nombre d'instructions élémentaires (création des tâches et des données) du langage L0/API effectuées ;
– N_{theft}, le nombre maximal de vols réussis par processeur.

La proposition suivante permet de relier T_p et N_{theft} aux coûts algorithmiques T_s, T_1 et T_∞ qui sont portables et indépendants du nombre de processeurs.

Proposition 1 *[49, 16] Avec une grande probabilité, le temps d'exécution T_p d'un programme avec l'algorithme de vol de travail sur p processeurs identiques est majoré par :*

$$T_p \leq T_1/p + c_\infty T_\infty \tag{4.1}$$

[2]Cependant, les résultats s'étendent à des choix déterministes, en particulier cycliques [101].

De plus,

$$T_1 \leq c_1 T_s \tag{4.2}$$

et le nombre de vols réussis par processeur est majoré par :

$$\max_{i=1,\ldots,p} (N_{theft})_i = O(T_\infty) \tag{4.3}$$

Les constantes c_1 et c_∞ permettent de quantifier le coût de l'implantation. Toute instruction supplémentaire exécutée à chaque instruction de l'application augmente la constante c_1. Toute instruction exécutée au moment des opérations de vol augmente la constante c_∞. Le principe du *travail d'abord* [85] s'énonce alors comme suit :

Principe 1 *[«Travail d'abord»] Pour la réalisation d'une fonctionnalité dans l'implantation, il convient de réduire les opérations qui seront effectuées systématiquement lors de l'exécution du programme afin de les reporter au moment des opérations de vols de travail.*

La constante c_1 est très importante car elle mesure le surcoût ajouté par l'implantation pour la gestion du parallélisme en plus du travail de l'application (voir l'équation 4.2). Dans l'implantation de Cilk [85], les auteurs montrent qu'il est possible de réaliser une implantation fine telle que pour de nombreuses applications $c_1 \simeq 1$ et donc $T_1 \simeq T_s$ qui veut dire que le surcoût introduit par le langage de programmation parallèle de niveau 0/API pour la gestion du parallélisme peut être rendu très faible.

Les constantes c_1 et c_∞, comme nous le verrons dans les chapitres suivants, jouent aussi un rôle très important pour la gestion de la tolérance aux pannes.

4.5 KAAPI : une instance du modèle

Dans cette section, nous présentons l'intergiciel KAAPI qui est une instance du modèle présenté précédemment et constitue le cadre expérimental de ce travail. Il s'agit d'un intergiciel qui permet la programmation d'une application parallèle et distribuée et qui fournit un moteur exécutif pouvant être utilisé comme machine cible pour des langages de plus haut niveau, comme ATHAPASCAN -1 [49]. Cette section présente en détail les deux niveaux de l'exécution d'un programme parallèle (§4.2).

4.5.1 KAAPI : Niveau 0/API

Le langage de programmation présenté ici est issu des travaux sur l'interface applicative ATHAPASCAN -1 [49, 53] et est une abstraction de l'interface proposée dans KAAPI . La description des programmes repose sur les concepts suivants : tâches, objets partagés et droit d'accès. Une tâche est un appel de procédure annoté dont le corps est une suite d'instructions comportant éventuellement des instructions de création de tâches. Une tâche peut être exécutée sans interruption et donc ordonnancée de manière non préemptive. Chaque donnée à laquelle la tâche accède est explicitement passé en paramètre à celle-ci lors de l'appel de procédure. Une tâche déclare les objets partagés auxquelles elle accède et spécifie pour chacun de ces objets le type d'accès (lecture ou écriture) qu'elle et ses descendantes effectueront.

4.5.1.1 Instructions du langage KAAPI /L0

Afin de décrire ce langage de programmation, deux objets (au sens objet dans les langages de programmation par objet) ont été définis : l'objet **tâche** qui permet de définir un ensemble d'instructions à exécuter et l'objet **donnée** qui à son tour permet de définir les données partagées ainsi que les modes d'accès à ces données.

Pour gérer et manipuler ces deux objets (tâche et donnée), deux instructions sont définies [49, 101] et ajoutées au langage de programmation séquentielle de base (C++).

Définition 5 *Le langage de programmation des applications parallèles dans le modèle considéré, appelé langage API ou L0, est constitué des instructions du langage C++ augmenté des instructions suivantes :*

- ***CreationDonnee** : cette instruction permet la création d'un objet donnée qui peut être partagé (en écriture ou en lecture) par plusieurs tâches s'exécutant sur plusieurs processeurs ;*
- ***CreationTache** : permet la création d'un objet tâche recevant en paramètre des objets données (en lecture ou en écriture) ou des valeurs immédiates.*

La figure 4.4 montre un exemple de programme en s'intéressant essentiellement aux instructions ajoutées à C++ définies ci-dessus.

```
1     CreationDonnee<double>  d1, d2, d3, d4, d5 ;
2     ...
3     CreationTache t1 ({Lecture, d1} , {Lecture, d2} ,
                        {Ecriture, d3}, {Ecriture, d4}) ;
4     CreationTache t2 ({Lecture, d3} , {Accumulation, d5}) ;
5     ...
6     CreationTache t3 ({Lecture, d4},  {Accumulation, d5}) ;
7     ...
```

Figure 4.4 – Exemple d'utilisation du langage.

Dans la ligne 1, on crée cinq objets partagés $d1, d2, d3, d4$ et $d5$ de type $double$. La création de la tâche $t1$, ligne 3, est faite par l'utilisation de l'instruction **CreationTache** en spécifiant pour chaque paramètre, le mode d'accès que $t1$ effectue à ce paramètre. La tâche $t2$, ligne 4, accède à la donnée partagée $d3$ en lecture et à $d5$ en écriture accumulée.

4.5.1.2 Sémantique du langage KAAPI /L0

La sémantique associée à ce langage de programmation est de type «séquentielle» : toute lecture voit la dernière écriture selon un ordre séquentiel total, lexicographique, associé au programme. Cet ordre est appelé *ordre de référence* et est défini ci-dessous ; il est très proche de l'ordre séquentiel en profondeur d'abord.

Définition 6 *L'ordre total sur les tâches $\prec_r$, dit **ordre de référence** est défini comme suit : supposons que t, t' et t_i sont des tâches,*

1. *Si t crée t' alors $t \prec_r t'$;*
2. *Si l'exécution de t crée les tâches $t_1, t_2, ..., t_n$ dans cet ordre alors $t \prec_r t_1 \prec_r t_2 \prec_r ... \prec_r t_n$;*
3. *Soient t_i et t_{i+1} deux tâches sœurs tel que $t_i \prec_r t_{i+1}$ alors pour toute tâche t' créée lors de l'exécution de la descendance de t_i, $t' \prec_r t_{i+1}$;*
4. *$\forall t$: la tâche principale (main du programme) $\prec_r t$;*

Le respect de l'ordre de référence a des conséquence sur la construction du graphe. Par exemple, lorsqu'une tâche est créée avec un accès en lecture sur une donnée, la valeur lue est celle produite par la ou les tâches qui la précèdent dans l'ordre de référence et qui prennent cette donnée en écriture.

4.5.1.3 Construction du graphe de flot de données : KAAPI /MV0

Les instructions du langage d'API de niveau 0 (KAAPI /L0) sont interprétées afin de construire la représentation sous forme d'un *graphe de flot de données* des tâches créées par le programme. Cette représentation est un élément clé du modèle de programmation présenté ici : que ce soit pour le calcul automatique des dépendances et d'un ordonnancement [49, 50, 53, 101] des tâches du programme ; ou pour la définition d'un état global permettant la sauvegarde et la reprise du calcul en cours [59, 61].

Le graphe de flot de données est construit dynamiquement à l'exécution du programme. Au début de l'exécution d'un programme, le graphe est initialisé avec la tâche principale initiale (le "main" du programme), qui démarre sur un seul processeur. L'exécution de chaque instruction du langage de programmation (présentée dans la définition 5) est associée à une opération de modification du graphe de flot de données. Lors de l'exécution d'une instruction «**CreationDonnee**», un nouveau nœud «donnée» est initialisé. Lorsqu' une instruction «**CreationTache**» est exécutée, un nouveau nœud «tâche» est inséré dans le graphe. Les arcs qui représentent les dépendances sont insérés dans le graphe en fonction des modes d'accès aux objets données partagées déclarés lors de la création d'une nouvelle tâche. Une fois que l'exécution d'une tâche se termine, le nœud tâche concerné est supprimé du graphe. Dès qu'il n'y a plus d'accès à un objet donnée, le nœud correspondant est supprimé du graphe.

La figure 4.5 illustre l'état du graphe de flot de données à différentes étapes de l'exécution du programme présenté dans la figure 4.4. Les parties a, b, c et d montrent l'état du graphe après l'exécution des lignes 1, 3, 4 et 6 du programme.

Cette construction est appelée «interprétation» et consiste à exécuter le code des tâches de la manière suivante : exécution séquentielle standard des instructions C++ ne contenant pas de création de tâches et interprétation des instructions de création de tâches et de données pour les ajouter dans le graphe. Le corps des tâches créées sera exécuté ultérieurement après ordonnancement.

4.5.2 KAAPI : Niveau 1/Ordonnancement

Nous venons de présenter les instructions du langage de niveau applicatif et la manière dont est construit la représentation abstraite de l'exécution sous forme d'un graphe de flot de données. Cette section présente le niveau 1 qui permet l'ordonnancement de ce graphe et son exécution distribuée par un algorithme de vol de travail.

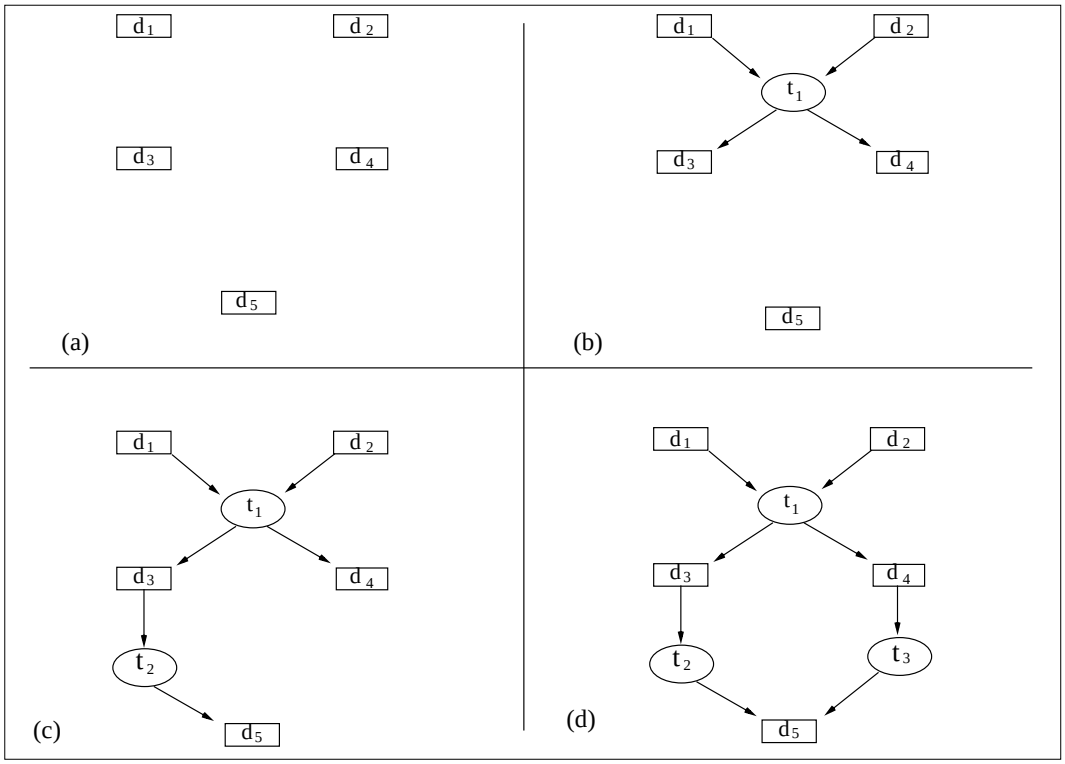

Figure 4.5 – Construction dynamique du graphe de flot de données.

4.5.2.1 Environnement d'exécution et langage KAAPI /L1

Les ressources du modèle peuvent être des ressources de calcul (processeurs), des ressources de stockage ou les deux. Un contrôleur (ou gestionnaire) de ressources existe et a la responsabilité d'affecter des ressources à l'application lorsque des ressources disponibles sont détectées, ou d'en retirer si nécessaire, et, enfin, de détecter et de signaler la présence des pannes apparaissant sur les ressources qu'il contrôle. Pour permettre l'ajout et le retrait de ressources, l'exécution de l'application est réalisée par un ensemble dynamique de processus appelés "exécuteurs" ; à chaque ressource est associée un ou plusieurs exécuteurs. Ces processus exécuteurs sont le cœur du modèle abstrait d'exécution. Une application s'exécute sur un ensemble de processus exécuteurs placés sur un ensemble de ressources. L'ajout de ressources à une application se traduit par la création d'un processus exécuteur pour exécuter le code de l'application.

La définition suivante précise cette notion de processus exécuteur en s'intéressant à l'état d'un tel processus dans KAAPI .

Définition 7 *Un processus "exécuteur" est associé à un flot de contrôle unique et est représenté à tout instant par son contexte d'exécution qui est décrit par :*

1. *des variables d'état qui caractérisent l'état local du processus séquentiel :*
   - *un nom unique dans le système représenté sous la forme d'un entier ;*
   - *l'état d'activité du processus (actif, inactif, bloqué, voleur, terminé) ;*
   - *l'état du flot de contrôle (registres) qui contient les informations nécessaires pour exécuter les instructions.*

2. *des variables d'état qui caractérisent les tâches qui sont placées sur ce processus exécuteur :*
   - *l'état de la tâche en cours d'exécution par le processus (pile C++ d'appel de fonctions) ;*
   - *le sous-graphe de flot de données des tâches de l'application créées par le processus (stocké dans une autre pile).*

Les processus "exécuteurs" du modèle d'éxécution ont la responsabilité d'évaluer le graphe de flot de données créé par l'exécution des instructions du langage L0/API (définition 5). Cette évaluation est calculée par un algorithme distribué d'ordonnancement. Cette évaluation est décrite par un programme écrit dans le langage de type L1 qui est C++ augmenté des instructions définies ci-dessous.

Définition 8 *Le langage de programmation de l'algorithme d'évaluation des graphes de flot de données, appelé langage* KAAPI */L1, est constitué des instructions d'un langage séquentiel (C++) augmenté des instructions suivantes :*

- ***ExportationTache** : cette instruction permet d'exporter (i.e. préparer et envoyer) une tâche vers un autre processus.*
- ***ImportationTache** : cette instruction permet d'importer (i.e. recevoir et préparer) une tâche exportée par un autre processus. Elle est exécutée par le processus recevant la tâche.*
- ***ExecutionTache** : qui permet d'exécuter le corps d'une tâche écrit dans le langage applicatif* KAAPI */L0 (définition 5) ;*

On note que les trois instructions précédentes sont au «niveau 1 » de la machine KAAPI , qui possède la représentation abstraite (graphe de flot de données) comme entrée. Ce graphe est implanté par une structure de données pouvant être parcourue.

Terminons cette section en précisant que les processus "exécuteurs" peuvent être préemptés sur réception d'un signal. Les processus peuvent définir le code d'interruption qui sera appelé pour traiter le signal.

4.5.2.2 Evaluation du graphe de flot de données : MV 1

Afin d'exécuter les tâches d'une application en respectant les dépendances décrites par le graphe de flot de données, les tâches prêtes (définition 9) à être exécutées doivent être connues à tout moment. Pour cela, un état est associé à chaque nœud tâche du graphe de flot de données. Le tableau 4.1, représente les changements possibles des états des noeuds tâches du graphe de flot de données.

Définition 9 *[«Tâche prête»]*

- *on dit que la donnée $d \in G$ en assignation unique est produite ssi toutes les tâches qui précèdent d dans G (selon l'ordre de référence) sont terminées ;*
- *on dit que le nœud tâche $t \in G$ est **prêt** ssi, $\forall$ l'arête $(d_i, t) \in G$ alors la donnée d_i a été produite.*

L'évaluation d'un graphe de flot de données consiste en l'exécution de tous les nœuds associés aux tâches qui sont prêtes dans le graphe. La sortie (résultats) de l'évaluation d'un

Etat	Tâche
C : créée	la tâche est créée, sa création est terminée.
P : prête	tous les paramètres auxquels on accède en lecture ont été produits
E : exécution	la tâche est en cours d'exécution
V : exportée (volée)	la tâche est exportée (volée)
T : terminée	l'exécution de la tâche est terminée
D : détruite	la tâche est détruite

Tableau 4.1 – États des noeuds tâches du graphe de flot de données

graphe de flot de données dépend uniquement de ses entrées (son état initial), mais pas de l'ordre d'exécution de ses tâches ou de sa forme. En conséquence, deux évaluations différentes d'un graphe avec des entrées identiques donnent les mêmes résultats, ainsi une évaluation du graphe à partir des entrées intermédiaires conduit aux mêmes résultats.

La figure 4.6 illustre l'évaluation (l'exécution) de graphe du programme de la figure 4.4 : 1) Initialement, la tâche t_1 est prête car ses paramètres en entrées i.e. les données d_1 et d_2 sont déjà produites. 2) t_1 peut alors s'exécuter, 3) Une fois que t_1 termine son exécution, elle produit les données d_3 et d_4 et donc les tâches t_2 et t_3 deviennent prêtes. 4) Ces tâches peuvent alors s'exécuter (éventuellement en parallèle). 5) la donnée résultat d_5 est produite lorsque t_2 et t_3 terminent leur exécution.

Comme le montre la figure 4.6, durant son évaluation, le graphe de flot de données représente le futur de l'exécution d'une application. Le résultat de l'évaluation de ce graphe d_5 peut tout à fait être produit à partir de l'état initial, les données d_1 et d_2, ou à partir d'un état intermédiaire par exemple les données d_3 et d_4.

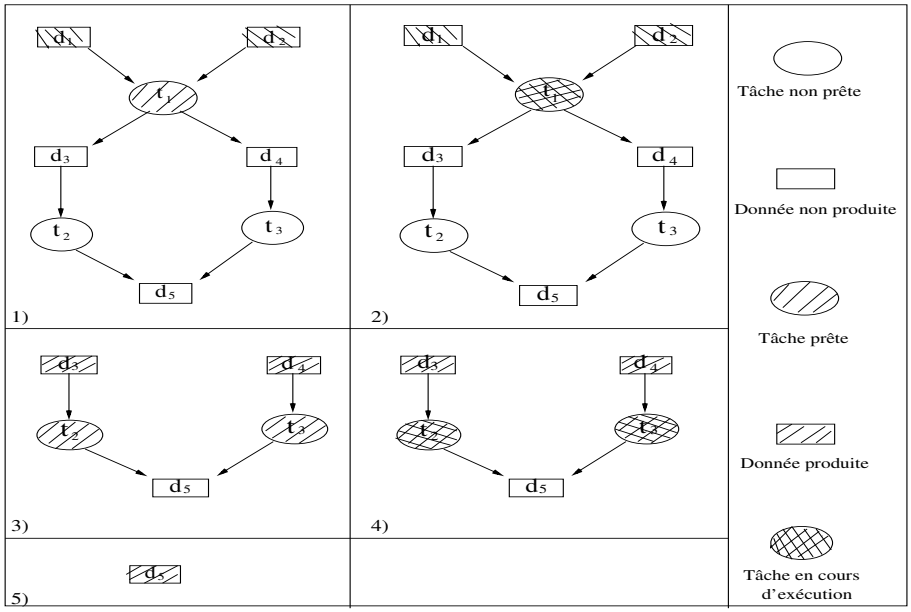

Figure 4.6 – Évaluation du graphe de flot de données selon les contraintes de flot.

4.5.2.3 Algorithme d'exécution distribué par vol de travail

Dans cette section nous illustrons l'utilisation des instructions du langage L1 (définition 8) pour l'écriture du programme de démarrage des processus du modèle d'exécution. Nous illustrons cet exemple avec l'implantation d'un ordonnancement distribué de vol de travail (§4.4).

```
0    // L'état du processus est voleur
     // Initialiser une nouvelle tâche le processus exécuteur.
1    ImportationTache ;
2    Tant que (il existe des tâches localement) {
3       Si (il existe une tâche prête) {
4         // L'état du processus est actif
5         ExecutionTache ;
6       }
7      Sinon {
8        // L'état du processus est inactif
9        Attendre qu'une tâche créée localement devienne prête
         ou qu'il ne reste plus de tâches localement.
11    }
13   } //Fin Tant que
14   // L'état du processus exécuteur est terminé.
```

Figure 4.7 – Code de démarrage de tous les processus du modèle d'exécution en mode «voleur de travail».

La figure 4.7 donne le code de démarrage d'un processus sur une ressource disponible. Lorsque un processus devient inactif, ligne 9, le système (KAAPI) crée et démarre un nouveau processus. Le code de traitement d'une requête de vol se fait sur interruption du processus victime qui reçoit la requête et est illustré dans la figure 4.8.

```
0   Si (signal exportation tâche) {
1      Si (il existe une tâche prête en locale) {
2            ExportationTache() ;
3      }
4   }
```

Figure 4.8 – Code d'interruption d'un processus qui reçoit une requête de vol.

La figure 4.9 illustre un exemple de graphe distribué de flot de données sur deux processeurs P_0 (victime) et P_1 (voleur) : le graphe devient distribué quand la tâche T_2 est exportée sur le processeur P_1. La tâche T_r est insérée dans le graphe et se charge d'envoyer au processeur P_0 la donnée r_2 produite par la tâche T_2 sur P_1 et de signaler la terminaison de la tâche T_2 sur P_0 afin de respecter les contraintes de flot (dépendances des données).

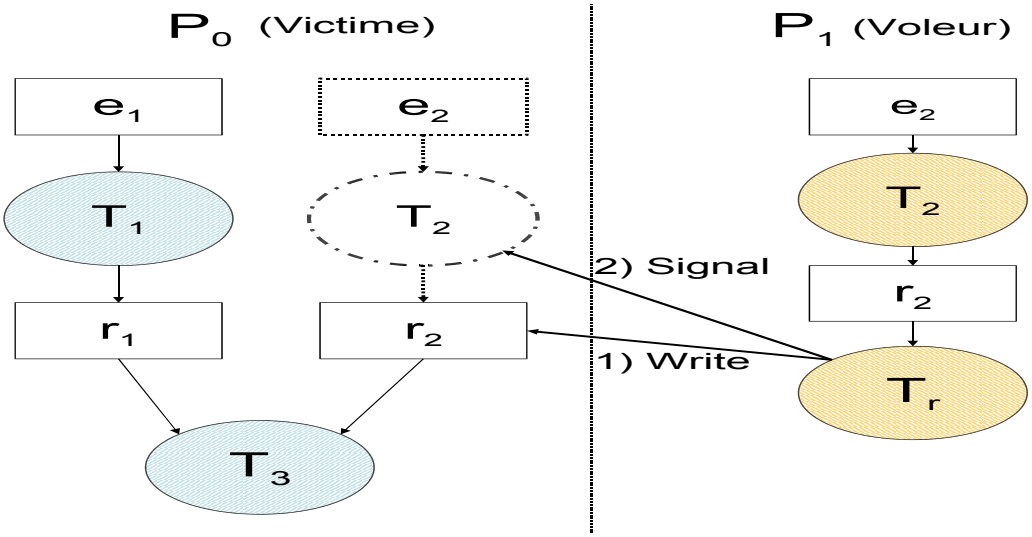

Figure 4.9 – Graphe distribué de flot de données

4.6 Conclusion

Dans ce chapitre, nous avons présenté le modèle de programmation et d'exécution d'applications parallèles cible de ce travail. Ce modèle est basé sur la représentation de l'exécution par un graphe de flot de données qui décrit les dépendances entre les tâches de l'application. L'intergiciel KAAPI qui est une instance du modèle de programmation et d'exécution considéré a été présenté.

Dans le cadre de KAAPI , la représentation abstraite sous forme d'un graphe de flot de données est close et causalement connectée à l'application. Dans le cadre des autres logiciels adoptant le modèle considéré, la représentation est toujours causalement connectée à l'application puisqu'elle sert à l'ordonnancement sur architecture multi-processeurs. Dans le cadre de Cilk [15] ou Satin [123], le calcul de la clôture de la représentation abstraite est non trivial et peut amener à considérer trop de données. Dans le cadre des logiciels Mentat [93], Orca [10] et Jade [102, 103] qui possèdent a priori une représentation abstraite close, il n'est pas clair qu'une partie de cette représentation ne repose pas sur des données internes représentant les flots d'exécution (registres du processeur et pointeur de pile) ce qui limiterait son utilisation à des architectures homogènes ou ce qui demanderait une étude approfondie de l'évaluation du graphe de flot de données pour déterminer les dates où la représentation abstraite représenterait correctement l'état du processus.

La modélisation de l'application parallèle par un graphe de flot de données a l'avantage, outre le fait qu'elle permet la gestion efficace du parallélisme indépendamment du nombre des ressources disponibles, de définir un état abstrait de l'exécution. Ceci est dû au fait que le graphe de flot de données qui modélise l'application contient uniquement les appels de fonctions (tâches) et leurs dépendances aux données. La définition et l'analyse des deux protocoles de tolérance aux pannes adaptés au modèle de programmation et d'exécution que nous l'avons présenté ici font l'objet du prochain chapitre.

Chapitre 5

Tolérance aux pannes et graphe de flot de données

Sommaire

5.1 Introduction

Comme nous l'avons présenté dans les chapitres précédents, la sauvegarde d'états intermédiaires d'un système parallèle réparti est le socle de la réalisation d'un protocole de tolérance aux pannes fondé sur l'utilisation d'une mémoire stable. Après une panne, l'état du composant défaillant peut être restauré à partir de son dernier état sauvegardé. L'hypothèse faite pour la mise en oeuvre d'un tel protocole est que l'on dispose d'une mémoire stable. L'état sauvegardé et le protocole de sauvegarde jouent un rôle très important pour déterminer les conditions et les critères dans lesquelles la reprise après panne est possible.

Dans ce chapitre, nous définissions et analysons deux protocoles (TIC : *Theft-Induced Checkpointing et* SEL *:* Systematic Event Logging) pour la tolérance aux pannes basés sur la sauvegarde d'un état abstrait de l'exécution. Ces deux protocoles ont pour objectif l'exécution fiable et efficace des applications parallèles réparties du modèle de programmation et d'exécution présenté dans le chapitre précédent. Ces deux protocoles de tolérance aux pannes ont été intégrés dans le logiciel KAAPI et font l'objet d'expérimentations dans les chapitres suivants.

Nous commençons ce chapitre par la présentation de ce qu'est l'état d'une exécution, en nous basant sur les définitions données dans le chapitre précédent. La définition de ce qu'est l'état d'une exécution est la différence la plus importante entre notre travail et les autres travaux présentés dans le chapitre 3. L'intérêt de notre représentation est de permettre la sauvegarde des points de reprise dans des environnements hétérogènes avec la possibilité d'effectuer la reprise quel que soit le nombre et le type des processeurs disponibles.

5.2 État d'exécution et définition des points de reprise

Pour définir l'état de l'exécution d'une application, nous utilisons la notion de graphe de flot de données, tel que présenté dans la définition 3 du chapitre précédent.

Ce graphe est une représentation abstraite du calcul à effectuer : les nœuds tâches représentent le calcul et les nœuds données partagées par les tâches qui représentent le flot courant des entrées aux sorties. Dans le cadre du modèle d'exécution considéré dans cette thèse et présenté dans le chapitre précédent (§4.5.2), ce graphe de flot de données G est dynamique : il change durant l'exécution du programme, par exemple, lors de l'exécution des tâches.

Bien que le graphe G soit vu comme un graphe global unique, sa mise en oeuvre peut être distribuée. Spécifiquement, chaque processus i contient et exécute un sous graphe G_i de G.

Une copie du graphe de flot de données G constitue un point de reprise global de l'application. Dans ce travail, un point de reprise concerne un processus, et se compose d'une copie de son graphe local G_i. Le protocole de sauvegarde doit s'assurer que les points de reprise sont créés d'une telle façon que G est toujours un état global cohérent de l'application et ceci même si un seul processus seulement est restauré.

Le point de reprise d'un processus i est son graphe local G_i, c.-à-d. ses tâches et leurs paramètres associés, et non pas l'état d'exécution des tâches sur le processeur lui-même. La compréhension de la différence entre ces deux concepts est cruciale. La sauvegarde de tâches et de leurs paramètres exige simplement le stockage de graphes de flots de données. La sauvegarde de l'exécution d'une tâche exige le stockage de l'état d'exécution du processus

qui est défini habituellement par le contexte du processeur c.-à-d. les registres de processeur tels que les compteurs de programme et les pointeurs de sa pile aussi bien que les données.

Dans le premier cas, il est possible de déplacer une tâche et ses entrées, en supposant que les deux sont représentées dans un mode ne dépendant pas de la plateforme. Dans le second, le fait que le contexte du processus dépende de la plateforme demande, afin d'exécuter une opération de restauration, soit un système homogène, soit la virtualisation de cet état [115].

L'intérêt de ce graphe dans le cadre du modèle d'exécution étudié ici est qu'il partage l'état d'un processus en deux parties distinctes :

1. la partie qui est dépendante de l'architecture : l'exécution du corps d'une tâche ;
2. la partie qui en est indépendante : les tâches à exécuter ainsi que leurs dépendances de données, décrites dans le graphe G.

De ce fait, le graphe ou des parties de celui-ci peuvent être déplacés à travers des plateformes durant l'exécution de l'application parallèle et répartie.

La représentation structurée du graphe G permet la souplesse de la reprise locale d'un processus P_i à partir de sa partie G_i de G ; dans le cas d'une seule défaillance, on peut ne pas exécuter de reprise globale. Cela est dû au fait que G_i, par définition du graphe de flot de données, contient toute l'information nécessaire pour identifier exactement les données qui sont absentes, ce qui inclut également l'information liée aux dépendances entre G_i et G_j, $i \neq j$.

Par conséquent, le moment où un point de reprise peut être réalisé se situe soit avant soit après l'exécution d'une tâche de l'application. Les points de reprise d'un graphe de flot de données ne contiennent que le futur de l'exécution, c.-à-d. les tâches à exécuter et les données nécessaires. Certaines données provisoires liées à l'exécution d'une tâche ne sont pas nécessaires pour le futur de l'exécution et ne sont pas donc sauvegardées ce qui résulte en la réduction de la taille des point des reprise.

5.3 Sauvegarde induite par le vol de travail (TIC)

Le protocole TIC que nous présentons dans cette section, peut tirer profit d'une formulation de l'état d'exécution permettant uniquement le recouvrement des processus défaillants. Nous noterons par G_i la représentation du graphe de flot de données pour le processus P_i, et par CP_i^j le j^e point de reprise du processus P_i. L'indice inférieur dénote le processus et l'indice supérieur l'instant du point de reprise.

La motivation du protocole TIC (*Theft-induced checkpointing*) est la méthode présentée dans [7] où la construction d'un état global cohérent est garantie lors de la sauvegarde des points de reprise en forçant la création des points de reprise lors de certaines opérations de communication. Dans le cadre du protocole TIC, les points de reprise sont forcés lors des opérations de vol de travail.

5.3.1 Protocole

Cette section commence par la description du protocole, illustré dans la figure 5.1, notamment en ce qui concerne les communications induites par les opérations de vol de travail. Dans le cadre du modèle d'exécution considéré, il s'agit de la seule cause de communication

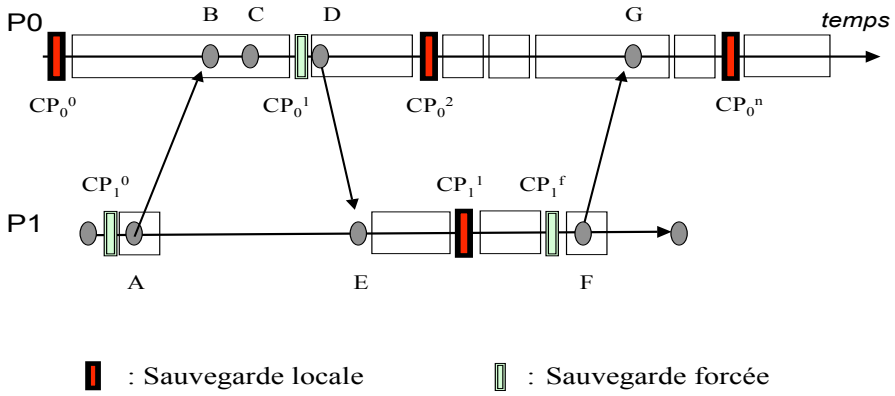

Figure 5.1 – Sauvegarde induite par le vol de travail : TIC .

(et ainsi de dépendances) entre les processus. Les points de reprise qui en résultent s'appellent *points de reprise forcés*. Nous considérons ensuite les points de reprise périodiques, appelés *points de reprise locaux* : il s'agit de ceux qui sont stockés régulièrement après l'expiration d'une période pré-définie qui sera notée τ.

5.3.1.1 Sauvegardes Forcées

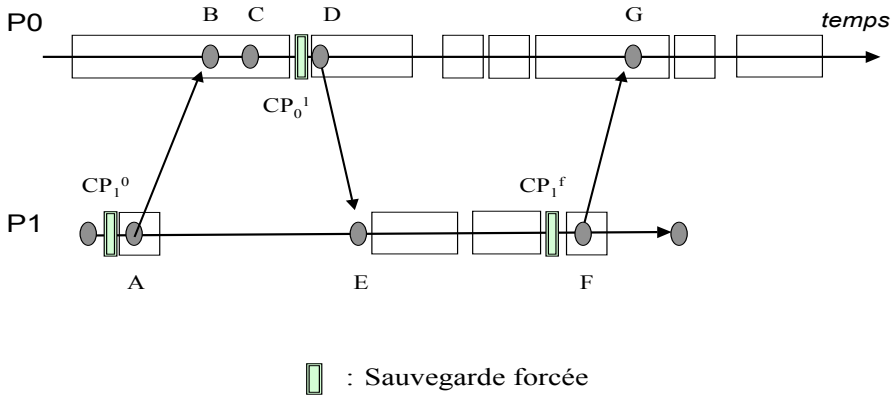

Figure 5.2 – Sauvegarde induite par le vol de travail TIC : sauvegardes forcées .

Les points de reprise forcés du protocole TIC sont représentés dans la figure 5.2. Les événements de A à G notés sur les deux processus P_0 et P_1 sont liés aux différentes opérations durant le traitement d'une opération de vol de travail. Initialement, le processus P_0 exécute une tâche. La séquence d'opérations suivante a lieu lorsque le processus P_1 commence une opération de vol.

- Un processus P_1 devient inactif. Il choisit alors un processus victime, ici le processus P_0, sur lequel il va effectuer une opération de vol. P_1 crée une «tâche de vol» t_t chargée d'effectuer le vol vers P_0. Avant d'exécuter cette tâche t_t, le processus de P_1 sauvegarde son état dans CP_1^0. L'événement A est l'exécution de t_t qui envoie une *requête de vol* à P_0.
- L'événement B est la réception de la «requête de vol» par P_0. L'événement C est la sélection d'une tâche prête t_s qui sera retournée à P_1 et qui est marquée comme étant «volée par P_1». Entre les événements B et C, le processus victime P_0 est dans une section critique vis à vis du vol.
- L'événement D constitue l'envoi de la tâche t_s à P_1. Avant cet événement D, P_0 force un point de reprise pour refléter le vol. Actuellement P_0 devient victime.
- L'événement E est la réception sur P_1 de la tâche volée. Le processus voleur P_1 crée deux tâches : la tâche volée t_s et une tâche t_r chargée de renvoyer les résultats de l'exécution de la tâche volée vers P_0. Cette dernière tâche devient prête lorsque la tâche t_s termine son exécution.
- Quand P_1 termine l'exécution de t_s, il sauvegarde un point de reprise stockant le résultat et puis exécute t_r qui renvoie le résultat à P_0 : il s'agit de l'événement F dans la figure.
- L'événement G est la réception du résultat par P_0.

Ces sauvegardes forcées sont complétées par des sauvegardes périodiques appelées «sauvegardes locales.»

5.3.1.2 Sauvegardes locales

Les points de reprise locaux pour chaque processus i, c.-à-d. G_i, sont stockés périodiquement, après l'expiration d'une période pré-définie τ. Plus précisément, après l'expiration d'un temps τ, un processus reçoit un signal pour sauvegarder un point de reprise.

Il y a, cependant, deux exceptions. D'abord, si le processus a une tâche dans l'état *exécution* il attend jusqu'à ce que l'exécution de cette tâche soit terminée. En second lieu, si un processus est dans une section critique ; par exemple entre les événements B et C dans la figure 5.2, le point de reprise est retardé jusqu'à la sortie de la section critique. Les points de reprise locaux sont illustrés dans la figure 5.3 pour les processus P_0 et P_1.

5.3.2 Reprise

Nous avons présenté l'ensemble des points de reprise qui sont sauvegardés au cours du protocole TIC lors de l'exécution des tâches par l'algorithme de vol de travail du modèle d'exécution. Dans cette section, nous présentons la restauration d'un processus après une défaillance : cette restauration suffit à masquer la défaillance.

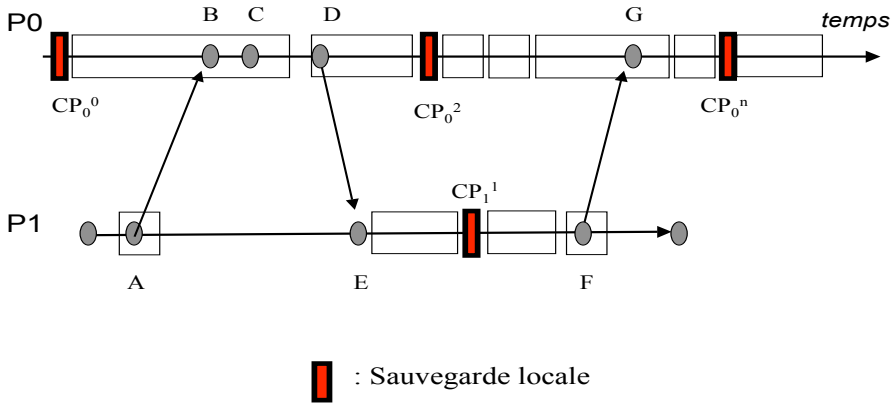

Figure 5.3 – Sauvegardes locales du protocole TIC : sauvegardes locales

5.3.2.1 Principe

L'objectif de TIC est de permettre la seule restauration des processus défaillants et donc d'éviter d'avoir à effectuer une restauration globale de l'application. La conséquence principale est la diminution du coût de la sauvegarde et de la restauration qui sera étudié dans les chapitres suivants.

Pour montrer que l'on peut restaurer uniquement le processus défaillant à partir de son dernier point de reprise, tout en garantissant un état global cohérent de l'exécution, on doit considérer les deux questions suivantes :

Q1 : Que fait un processus qui a besoin d'envoyer un message à un processus défaillant ?

Q2 : Comment un processus restaurant un processus défaillant peut-il récupérer les messages qui ont été envoyés au processus défaillant après son dernier point de reprise et avant la détection de sa défaillance ?

5.3.2.2 Preuve de correction du protocole

En ce qui concerne la première question Q1, il faut noter que les seules communications vers un processus défaillant sont : l'émission d'une requête de vol ; l'émission d'une tâche suite à une requête de vol ; le retour des résultats suite à l'exécution d'une tâche volée. Dans le cas d'une émission d'une requête de vol, cela n'a pas d'impact sur l'exécution, ni sur l'état globale de l'application. En cas d'échec, la requête est considérée comme ayant échoué et le processus émetteur va ré-émettre une requête vers un autre processeur. En cas d'échec de l'émission d'une tâche suite à une requête de vol, l'opération de vol est annulée sur le processus victime ainsi que le point de reprise associé.

Dans le cas du retour des résultats suite à un vol, le processus qui émet le message contenant les résultats effectue une sauvegarde de ces résultats avant émission (sauvegarde forcée juste avant l'événement F dans la figure 5.2). Le message n'ayant pas abouti est donc

sauvegardé et sera rejoué ultérieurement lors de la restauration du processus défaillant : ce qui répond à la question Q1.

En ce qui concerne la deuxième question Q2, les seuls messages reçus par un processus sont :

1. une *requête de vol* (l'événement B) ;
2. la réception d'une tâche volée (l'événement E) ;
3. le résultat de la tâche volée (l'événement G).

Nous utilisons les événements numérotés dans la figure 5.1 pour traiter chacun de ces trois cas :

- **premier cas :** la perte d'une *requête de vol* (l'événement B) n'a aucune conséquence, soit il détecte l'erreur de sa communication, soit le voleur attend une réponse pendant un délai de garde et fera une autre demande si le délai de garde arrive à expiration.
- **deuxième cas :** si le voleur P_1 défaille après la réception de la tâche volée (l'événement E), mais avant qu'il soit en mesure de sauvegarder son état, il est simplement restauré comme P_1' à partir du point de reprise au point initial CP_1^0 où il re-demande une tâche de P_0 (l'événement A). Le processus victime P_0 identifie donc une requête redondante et il change l'état de la tâche marquée volée t_s en une tâche d'état *prêt*. La première requête de vol de P_1 vers P_0 est annulée. La seconde requête de vol de P_1' sera traitée par P_0 comme une nouvelle requête de vol.
- **troisième cas :** dans le cas d'une défaillance de la victime P_0 après qu'il ait reçu le résultat (l'événement G) mais avant qu'il ait pu sauvegardé son état, le processus victime est restauré sur un processus P_0' à partir d'un état où la tâche est encore marquée comme étant volée. Dans ce cas, P_0' analyse l'ensemble des tâches de son point de reprise. S'il trouve une tâche volée alors il demande au voleur de renvoyer les résultats à P_0' à la place de P_0. Cela se traduit soit par la ré-exécution de la tâche résultat t_r qui émet le résultat, si celle-ci est déjà terminée (donc sauvegardée avant l'événement F dans la figure 5.1). Soit, si le voleur exécute encore la tâche volée, par une indication du changement du destinataire du résultat (P_0' au lieu de P_0). Ainsi, la reprise de P_0 utilise G_0 et G_1, qui contient seulement t_r.

En répondant aux questions Q1 et Q2, nous avons prouvé qu'un état global incohérent ne peut pas se produire au moment d'une reprise. Cependant, il reste à établir pourquoi les trois points de reprise forcés montrés dans la figure 5.1 sont nécessaires pour la résolution des problèmes posés par Q1 et Q2. Notons CP_1^0 et CP_1^f respectivement le premier et le dernier point de reprise d'un voleur P_1.

Le point de reprise initial CP_1^0 garantit qu'il existe au moins une sauvegarde de *requête de vol* pour un voleur qui défaille. Ainsi, en cas de défaillance, le voleur est restauré sur un nouveau processus P_1'. Dans le plus mauvais cas, l'état de P_1 est restauré à partir du point de reprise initial et P_1' entrera en contact avec P_0 par une *requête de vol* contenant le vieil identifiant de processus. Sans CP_1^0 une défaillance avant la sauvegarde de l'état du processus voleur aurait entraîné le blocage de la victime puisque cette dernière n'aurait jamais reçu le résultat de la tâche volée.

Le point de reprise final CP_1^f est nécessaire au cas où la victime P_0 défaille après qu'elle ait reçu du voleur les résultats de la tâche volée, mais avant qu'elle ait pu sauvegardé son état reflétant ces résultats. Ainsi, si la victime défaille entre l'événement G et son premier point de reprise après G, les actions décrites dans l'étude de la réponse à Q2 assureront que la victime peut recevoir les résultats de la tâche volée.

Il est important de noter que le point de reprise final du processus voleur ne peut pas être supprimé jusqu'à ce que la victime ait pris un point de reprise après l'événement G. Ce point de reprise contient forcément les résultats de la tâche volée. Finalement, le point de reprise forcé du processus victime (entre les événements C et D) assure qu'une défaillance après ce point de reprise n'a pas comme conséquence la perte du calcul du processus voleur.

De manière similaire, on prouve que les actions liées aux deux questions Q1 et Q2 peuvent être vérifiées par énumération de tous les scénarios de défaillance possibles de la victime et du voleur, y compris les défaillances simultanées.

5.3.3 Conclusion

Dans cette section, nous avons vu le protocole TIC qui repose sur le modèle d'exécution présenté dans le chapitre précédent et plus particulièrement sur les opérations d'ordonnancement de l'application définies dans le langage de niveau 1 : les opérations de vols de travail (*work stealing*) dans notre contexte. La reprise après défaillance dans TIC est la restauration du dernier état sauvegardé du processus défaillant (P_i) et consiste à re-exécuter le sous graphe de flot de données G_i associé à P_i en suivant les actions décrites ci-dessus.

Le temps de calcul perdu suite à une panne avec le protocole TIC est généralement estimé égal à la période de sauvegarde τ (voir §5.5) qui peut être importante. Par conséquent, il est intéressant de limiter le temps de calcul perdu à la durée de l'exécution de la tâche courante au moment de la panne sur le processus défaillant[1].

Cette idée fait l'objet du second protocole SEL basé sur la journalisation des événements de construction du graphe de flot de données représentant l'exécution de l'application. Cela se traduit par un protocole qui repose sur les opérations de description des tâches constituant les opérations du langage de niveau 0 (L0) présenté dans le chapitre précédent.

5.4 Journalisation des événements du graphe (SEL)

Nous avons montré que le graphe de flot de données G est une vue de l'état d'exécution d'une application parallèle dans notre modèle d'exécution. Ce graphe G peut être considéré comme un état global de l'application. En pratique ce graphe G est distribué sur tous les processus : chaque processus i contient et exécute un sous-graphe G_i du graphe G.

Le graphe de flot de données à un instant t de l'exécution parallèle distribuée d'une application est un état global de l'exécution qui représente le calcul qui reste à effectuer, *i.e.* le futur de l'exécution. Par conséquent, si l'on possède une copie de ce graphe sur un support stable durant l'exécution, alors une reprise de l'exécution après une défaillance est possible à partir de cette copie.

Cette copie du graphe peut être réalisée en sauvegardant les opérations qui le modifient : il s'agit ni plus ni moins que de journaliser sur un support stable les événements de sa construction. La restauration d'un processus défaillant après panne consiste à rejouer ces événements afin de reconstruire l'état final décrit juste avant la panne. Pour que cela soit possible, il est nécessaire que le graphe représentant l'exécution parallèle d'une application soit unique : deux exécutions parallèles de l'application avec les mêmes entrées doivent

[1] A noter qu'il s'agit de l'unité de base de calcul dans la représentation sous forme de graphe de flot de données de notre modèle d'exécution.

construire le même graphe. Autrement dit, la construction et l'exécution du graphe doivent être déterministes (voir §3.6).

Avant de décrire le protocole SEL, nous définissons d'abord les hypothèses à faire sur le graphe de flot de données et son exécution afin de garantir une exécution déterministe de l'application.

5.4.1 Hypothèses d'exécution déterministe

Dans cette section, nous présentons les hypothèses qui doivent être faites sur les tâches du graphe de flot de données pour pouvoir utiliser un mécanisme de journalisation afin de réaliser la tolérance aux pannes dans notre modèle (voir §4.5.2). Ces hypothèses doivent garantir que les processus qui exécutent ce graphe respectent l'hypothèse PWD définie dans § 3.4.

L'ensemble de ces hypothèses est donné dans la définition suivante.

Définition 10 *Condition d'application de SEL.*

- H_1 *: toutes les synchronisations entre les tâches sont explicitement décrites dans le graphe de flot de données.*
- H_2 *: une tâche s'exécute jusqu'à la fin de son exécution sans synchronisation : une fois dans l'état prête, une tâche peut s'exécuter de manière non-préemptive sans attendre aucun des résultats des tâches filles créées.*
- H_3 *: les tâches sont déterministes : n'importe quelle exécution d'une tâche avec les mêmes entrées fournit les mêmes résultats. Les résultats d'une tâche peuvent être des tâches créées ou des données.*

Propriété 1 *Si les tâches du graphe de flot de données d'une application parallèle vérifient les hypothèses H_1, H_2 et H_3, alors tous les processus qui exécutent ce graphe respectent l'hypothèse PWD (voir §3.4).*

Preuve: L'exécution de l'application est décrite par le graphe de flot de données qui contient toutes les synchronisations entre les tâches de l'application (H_1). Chaque processus dans le modèle considéré exécute une ou plusieurs tâches de ce graphe. L'exécution d'une tâche est non-préemptive (H_2) et donc une fois qu'un processus démarre l'exécution d'une tâche, il n'a plus besoin d'aucune information externe pour terminer l'exécution de cette tâche. L'exécution de n'importe quelle tâche est déterministe (H_3) et ne dépend que de ses entrées. Par conséquent, l'exécution d'une tâche est indépendante des conditions temporelles et/ou spatiales de l'exécution du programme. Donc, l'état de chaque processus est modélisé par une séquence d'intervalles d'état : chaque intervalle représente l'exécution d'une tâche qui est déterministe et ne dépend que des entrées de la tâche. Puisque, le début de l'exécution d'une tâche nécessite que toutes ses entrées soient déjà produites (éventuellement produites à distance par d'autres processus), alors le début de l'exécution d'une tâche est un événement non-déterminite qui peut être identifié. Par conséquent, l'exécution de chaque processus est modélisée par une séquence d'intervalles d'état chacun débutant par un événement non-déterministe, ce qui correspond à l'hypothèse PWD. □

Il faut noter que les hypothèses H_1 et H_2 sont toujours vérifiées dans le modèle de programmation et d'exécution considéré (voir chapitre 4). L'hypothèse H_3 est suffisante pour garantir une exécution sous l'hypothèse PWD.

5.4.2 Journalisation du graphe

Dans cette section, nous définissons le protocole SEL qui s'appuie sur la propriété (1) précédente. Ce protocole est dérivé de la méthode de journalisation présentée dans [73]. Cette section débute par une description du principe du protocole, puis nous présentons en détail le protocole de journalisation sur support stable de stockage nommé SEL.

5.4.2.1 Principe

Sous les hypothèses H_1, H_2 et H_3, deux exécutions différentes d'une application, avec les mêmes entrées, construisent le même graphe de flot de données. Par conséquent, les nœuds du graphe peuvent être identifiés de manière unique et reproductible. Le stockage de l'identification et l'information associée à chaque nœud dans le graphe permet, suite à une exécution non terminée *i.e.* avec pannes, d'identifier les nœuds du graphe qui sont déjà terminés et ce qui reste à faire.

Le mécanisme de journalisation proposé (voir figure 5.4) est donc basé sur la propriété 1. Il consiste dans le stockage systématique sur un support stable de chaque tâche du graphe de flot de données (identification et paramètres) et de leurs dépendances vis-à-vis du flot de données (identification et valeur relative des dépendances).

D'un point de vue théorique, cette technique de journalisation évite l'*effet domino*. Par exemple, suite à une défaillance, le programme n'est jamais restauré à partir de son état initial. En effet, si le $MTBF$ est plus grand que la période maximale d'exécution d'une tâche ; alors on assure qu'au moins une tâche a été terminée avec succès avant la défaillance et qu'elle ne sera pas re-exécutée durant la phase de reprise.

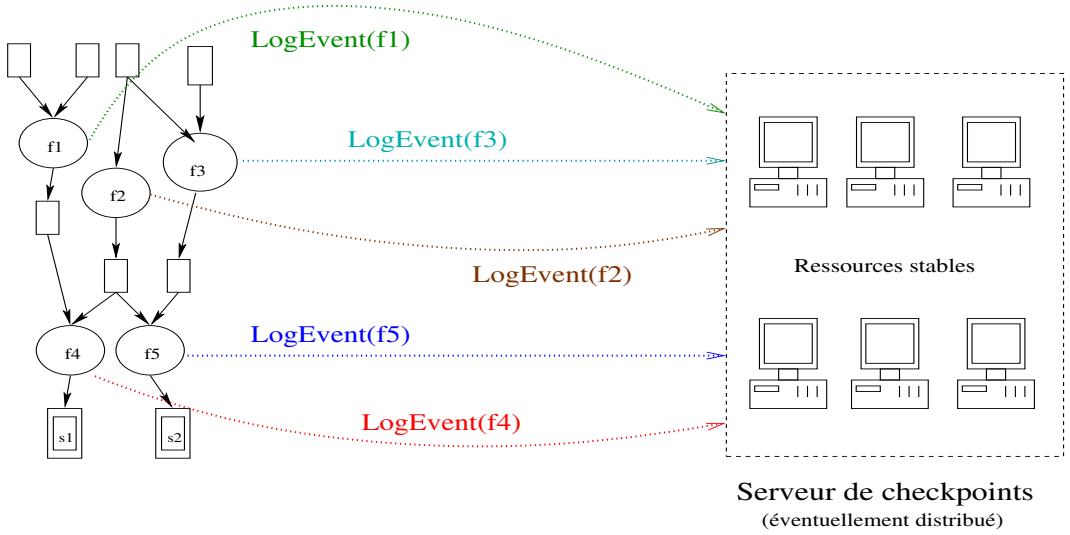

Figure 5.4 – Journalisation distribuée du graphe de flot de données

5.4.2.2 Identification des nœuds du graphe

Les identifiants des nœuds du graphe doivent être uniques et reproductibles : l'identifiant d'un nœud du graphe recréé après la défaillance d'un processeur doit être le même que son

identifiant avant la défaillance. Le choix de la méthode de génération des identifiants des nœuds dépend de l'application, *i.e.* de la forme du graphe. On utilise souvent le fait que tout nœud du graphe est créé par un unique prédecésseur : le graphe peut être énuméré en utilisant l'arbre de création associé.

Par exemple, si un majorant M du nombre de nœuds fils créés par tout nœud est connu à l'avance l'identification des nœuds se fait de la manière suivante [81, 128] : La tâche principale (main du programme) est la tâche qui commence le calcul et est la première tâche dans le graphe ; cette tâche a l'identifiant $Id = 1$. L'identifiant d'un nœud fils est calculé à partir de l'identifiant de sa tâche mère en multipliant l'identifiant de la mère par M et en lui ajoutant le nombre de nœuds créés avant lui par sa tâche mère. Par exemple, l'identifiant du troisième nœud fils de la tâche qui a l'identifiant $Id = 9$ en sachant que le nombre maximal de nœuds créées par une tâche est $M = 5$ peut être calculer comme suit : $9 * 5 + 3 = 48$.

Mais si le majorant M n'est pas connu, l'identifiant d'un nœud n dans le graphe peut être construit de la manière suivante : le nœud n est identifié par une paire contenant l'identifiant du nœud mère de n et l'ordre du fils n par rapport à sa mère.

Quel que soit la méthode choisie pour identifier les nœuds du graphe, pour les nœuds données l'identifiant est un couple de deux attributs ; le premier attribut est l'identifiant de nœud et le deuxième est le numéro de version de cette donnée. A la création d'un nœud donnée, son numéro de version vaut 0. Ce numéro sera augmenté à chaque modification de la donnée associée au nœud.

5.4.2.3 Protocole

Le protocole de journalisation SEL est basé sur l'automate d'état d'une tâche dans le graphe de flot de données. La figure 5.5 présente cet automate ; en effet, après le début et avant la fin de l'exécution, l'état actuel d'une tâche t est directement lié au nombre des tâches descendantes (filles) qu'elle a créées.

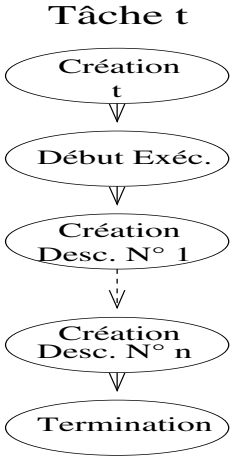

Figure 5.5 – L'automate d'état d'une tâche t.

Les informations nécessaires à la restauration d'une tâche dans le graphe sont la fonction qui représente le corps de la tâche et ses paramètres. Selon la propriété 1, les événements non-déterministes dont nous avons besoin afin de restaurer un processus défaillant sont les

informations concernant le début d'exécution d'une tâche et le fait que tous ses paramètres soient déjà produits.

Le principe de l'identification des nœuds est le suivant :

– **Pour les nœuds données "d" :**

1. à la création de d, une identification et un numéro de version sont associés au nœud d (voir §5.4.2.2) ;
2. l'identification, le numéro de version et la valeur associés à d sont sauvegardés sur un support stable ;
3. à chaque modification de la valeur de d, la mise à jour du numéro de version et de la valeur sont répercutés sur le support stable.

– **Pour les nœuds tâches "t" :**

1. à la création d'un nœud tâche t :
   – une identification est associée à la tâche ;
   – l'identification de t, le nom de la procédure associée à t, les paramètres de t et l'état de t sont stockés sur support stable.

   Parmi les paramètres de t, certains peuvent être des paramètres passés par valeurs et d'autres des références vers des versions de nœuds données avec leurs modes d'accès. Les paramètres passés par valeur sont stockés directement avec la tâche t. Les références sont remplacées par leur identifiants.
2. les changements d'état de la tâche t (présentés dans la figure 5.5) sont propagés au support stable.

5.4.3 Reprise

Comme pour le protocole TIC, l'objectif principal du protocole SEL est de pouvoir restaurer uniquement les processus défaillants sans devoir restaurer l'ensemble des processus de l'application. Ce protocole permet aussi de diminuer le temps de calcul perdu suite à une panne du fait qu'il est capable de restaurer chaque tâche exécutée.

5.4.3.1 Principe

Un processus qui doit être restauré, le fait à partir de son (fichier de) journal stocké sur un support stable. Pour montrer la possibilité de la reprise locale d'un processus à partir de son fichier de journal tout en garantissant un état global cohérent de l'exécution, il faut répondre aux trois questions suivantes :

Q1 : Que fait un processus qui a besoin d'envoyer un message à un processus défaillant ?

Q2 : Comment un processus restaurant un processus défaillant peut-il récupérer les messages envoyés au processus défaillant après son dernier point de reprise et avant la détection de sa défaillance ?

Q3 : Comment un processus restaurant un processus défaillant peut-il reconstruire le sous graphe de flot de données du processus défaillant en garantissant la cohérence globale du graphe de flot de données représentant l'exécution ?

Les deux questions Q1 et Q2 ont été déjà considérées pour le TIC (voir 5.3.2.2) et le même principe peut être appliqué pour le protocole SEL.

5.4.3.2 Protocole

Pour répondre à la troisième question Q3, nous décrivons par la suite un procédé de reconstruction du sous graphe perdu suite à une panne.

Puisque les événements de construction et de modification du graphe sont enregistrés sur le support stable de manière incrémentale, en cas de défaillance d'un processus, son journal contient le dernier état enregistré pour toutes les tâches créées sur ce processus. En effet, après le début et avant la fin de l'exécution, l'état actuel d'une tâche t est directement lié au nombre des tâches descendantes qu'elle a créées. Ces créations sont enregistrées avec succès sur le support stable.

Le mécanisme de recouvrement est dérivé de l'automate de la figure 5.5. Une tâche non terminée est restaurée à partir de son dernier état valide, enregistré avant panne. La figure 5.6 montre l'automate de restauration des tâches d'un processus défaillant.

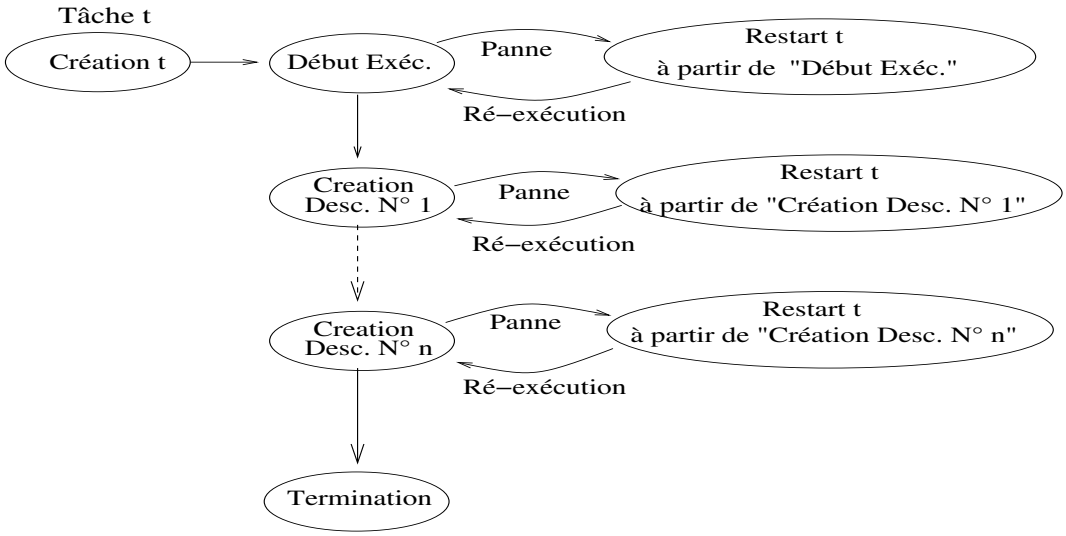

Figure 5.6 – L'automate de restauration d'une tâche après une panne.

Le protocole de reprise est le suivant. Une fois que la défaillance d'un nœud (P_i) est détectée, un nouveau processus (P_i') est créé pour restaurer P_i. Ce processus est lié au (fichier de) journal de P_i. Ce journal est lu et interprété comme cela a été présenté précédemment.

5.4.4 Conclusion

Sous les hypothèses H_1, H_2 et H_3 (voir §5.4.1) qui assurent des exécutions et ré-exécutions déterministes du programme, le mécanisme de journalisation et de reprise du protocole SEL vérifie les propriétés suivantes lors d'une exécution avec pannes :

1. un événement de construction du graphe de flot de données est stocké une seule fois sur le support stable ;
2. chaque tâche termine correctement son exécution une seule fois.

Puisque le mécanisme de journalisation pessimiste et de reprise proposé évite l'*effet domino*, le protocole de reprise assure qu'une exécution complète de l'application est atteinte après un nombre fini de ré-exécutions.

L'efficacité d'un protocole de tolérance aux pannes par sauvegarde / reprise dépend de plusieurs paramètres comme : le type d'application considérée, la machine d'exécution, le choix et la configuration du support stable et le niveau de disponibilité demandé. L'ensemble de ces paramètres permet de mesurer le surcoût introduit par le mécanisme de tolérance aux pannes pour une configuration donnée du matériel et du logiciel (application). Dans la section suivante, nous proposons une analyse théorique de coût des deux protocoles présentés afin de définir leurs avantages respectifs et leurs domaines d'application.

5.5 Analyse de complexité pour la tolérance aux pannes

Dans cette section, nous présentons une analyse de complexité pour les deux protocoles de tolérance aux pannes proposés. Cette analyse repose sur le modèle de coût (§4.4) associé à l'exécution d'une application parallèle dans le modèle de programmation et d'exécution considéré.

Dans l'analyse de coût associée aux protocoles TIC et SEL, nous faisons la différence entre les exécutions sans et avec défaillances. En outre, nous supposons que l'application possède un degré de parallélisme important, en reprenant les notations de la section §4.4, cela s'exprime par :

$$\frac{T_1}{p} \gg c_\infty T_\infty \tag{5.1}$$

Cette hypothèse est désignée sous le nom *"parallel slackness assumption"* dans [85]. En la présence d'un algorithme d'ordonnancement de type de vol de travail et pour la classe des programmes considérés avec notre modèle, cette hypothèse permet de considérer que l'exécution s'effectue avec un facteur d'accélération (*speedup*) quasi-linéaire : $T_p \approx \frac{T_1}{p}$.

5.5.1 Exécution sans panne

Si on ajoute un mécanisme de sauvegarde pour supporter la tolérance aux pannes, il est très intéressant d'analyser le surcoût associé à une exécution correcte (sans panne), puisque l'occurrence de défaillances est considérée comme l'exception plutôt que la norme.

5.5.1.1 Analyse du SEL

En utilisant le protocole SEL, pour chaque création d'un nœud du graphe, une sauvegarde des informations associées à ce nœud sur un support stable est effectuée. Donc, le coût de journalisation dépend la taille du graphe de flot de donnée G.

Notons par T_P^{SEL}, le temps d'exécution d'un programme KAAPI sur une machine composée de p processeurs, avec le protocole SEL. Alors,

$$T_P^{SEL} \leq \frac{T_1^{SEL}}{p} + c_\infty T_\infty^{SEL}. \tag{5.2}$$

T_∞^{SEL} désigne le temps d'exécution du chemin critique dans le graphe G si celui-ci s'exécute avec le protocole SEL ; on peut supposer $T_\infty^{SEL} = O(T_\infty)$.

T_1^{SEL} note le temps d'exécution séquentiel du programme avec la journalisation de son graphe par le protocole SEL, *i.e.* $T_1^{SEL} = T_1 +$ *coût de journalisation*.

Le coût de journalisation est une fonction de deux paramètres. Le premier dépend de la taille du graphe de flot de données du programme G, plus précisément du nombre de tâches et de leurs données. Le second dépend du temps d'accès élémentaire à la mémoire stable (serveur des points de reprise) ; ce paramètre est noté par t_s. Par conséquent,

$$T_1^{SEL} = T_1 + f_{overhead}^{SEL}(|G|, t_s). \tag{5.3}$$

La mesure réelle du coût de SEL est alors $T_1^{SEL} - T_1$, ce qui permet d'en déduire le coût de journalisation dans l'exécution parallèle, i.e. $T_P^{SEL} - T_p$.

5.5.1.2 Analyse du TIC

Dans notre protocole de sauvegarde induite par le vol de travail TIC, un point de reprise est pris périodiquement pour chaque processus, à l'expiration d'une période τ, et de plus est forcé sur chaque opération de vol de travail.

Notons par T_P^{TIC}, le temps d'exécution d'un programme KAAPI sur une machine composée de p processus, avec le protocole TIC. Alors, en supposant que les points de reprise de deux processus peuvent être enregistrés en parallèle,

$$T_P^{TIC} \leq T_p + \max_{i=1,\ldots,p} \{CheckpointOverhead_i\} \tag{5.4}$$

où $CheckpointOverhead_i$ note le surcoût total de la sauvegarde sur le processus i. Ce surcoût dépend d'une part, du nombre total des points de reprise réalisés par chacun des processus i et d'autre part, du surcoût dû à la réalisation d'un seul point de reprise.

Le nombre maximum de points de reprise réalisés par un processus est majoré par $[T_P^{TIC}/\tau + N_{theft}]$, où T_P^{TIC}/τ donne le nombre des points de reprise dus à la période de sauvegarde et N_{theft} est le nombre maximum de vols réalisés par n'importe quel processus.

Le surcoût d'un point de reprise dans le protocole TIC est différent de son surcoût dans le protocole SEL : avec TIC une sauvegarde est composée d'une collection de tâches dans G_i et non plus d'un événement de modification de l'état du graphe comme dans SEL.

A chaque instant et du fait de l'exécution locale «en profondeur d'abord», le nombre de tâches dans un sous graphe G_i est majoré par N_∞, où N_∞ représente le nombre maximum de tâches sur un plus long chemin dans le graphe G [85, 101]. Le surcoût de la sauvegarde est donc borné par :

$$\max_{i=1,\ldots,p} \{CheckpointOverhead_i\} \leq [T_P^{TIC}/\tau + N_{theft}]\ f_{overhead}^{TIC}(N_\infty, t_s) \tag{5.5}$$

La fonction $f_{overhead}^{TIC}()$ indique le surcoût associé à la réalisation d'un point de reprise. Ce surcoût dépend uniquement du graphe de flot de données de l'application G. Plus précisément, le surcoût dépend (avec une grande probabilité) de N_∞ et du temps élémentaire t_s nécessaire pour accéder au support stable de stockage.

Par conséquent, le surcoût de la sauvegarde avec le protocole TIC est :

$$T_P^{TIC} \leq T_p + [T_P^{TIC}/\tau + N_{theft}]\ f_{overhead}^{TIC}(N_\infty, t_s) \tag{5.6}$$

Sous l'hypothèse *"parallel slackness assumption"*[85], il est important de noter que le nombre de vols, N_{theft}, ayant pour résultat les points de reprise forcés, est borné et petit pour beaucoup d'applications [85, 101]. Donc, en choisissant une période τ appropriée le nombre de points de reprise locaux peut être ajusté afin d'obtenir $T_P^{TIC} \approx T_p$.

5.5.2 Exécution avec pannes

Le surcoût ajouté à une exécution correcte (sans pannes) est la pénalité qu'on paye pour avoir un mécanisme de recouvrement après pannes. Il reste à étudier le coût associé au recouvrement suite à une panne et le temps d'exécution qui a été perdu.

5.5.2.1 Analyse du SEL

Le coût associé au recouvrement est dû d'une part, au chargement des événements valides du graphe G à partir de la mémoire stable et d'autre part à la reconstruction de G.

La récupération d'événements de G à partir de la mémoire stable concerne uniquement les sous-graphes des processus affectés par les pannes. Donc, le temps de chargement à partir de la mémoire stable dépend la taille de G_i et ceci est dominé par la taille des données représentant les entrées des tâches dans G_i.

Donc, le temps de recouvrement d'un processus défaillant i, noté par $t^i_{recovery}$, dépend uniquement de la taille de sous-graphe G_i associé. Par conséquent, $t^i_{recovery}$ est de l'ordre de la taille du sous-graphe G_i, i.e. $t^i_{recovery} = O(|G_i|)$.

Il est important à noter que pour une reprise globale, qui est le résultat de la défaillance de tous les processus de l'application, le coût associé est $max(t^i_{recovery})$ et non pas $\sum t^i_{recovery}$.

L'avantage du protocole SEL est que, en raison de sa fine granularité, la quantité maximum de temps d'exécution perdu suite à une panne est celle d'une tâche. En outre, la reprise exige seulement le recouvrement d'une seule tâche et donc le coût associé est très faible. Cependant, le coût de la journalisation est élevé, comme illustré dans l'équation 5.3.

5.5.2.2 Analyse du TIC

Avec le protocole TIC, la quantité maximum de temps d'exécution perdu est généralement supérieur à la quantité perdue avec le protocole SEL. En effet, après une défaillance, la quantité de travail qu'un processus peut perdre est la durée d'exécution entre deux points de reprise consécutifs. Cette durée est définie par la période de sauvegarde τ et le temps maximum d'exécution d'une tâche, puisqu'un point de reprise pour un processus n'est possible que lorsqu'il n'y a pas de tâche en cours d'exécution.

Dans le plus mauvais de cas, le processus reçoit un signal de sauvegarde après τ et doit attendre la fin de l'exécution de sa tâche courante avant de sauvegarder son état. Ainsi, le temps entre les points de reprise est borné par $\tau + \max(d_i)$ où d_i est la durée d'exécution d'une tâche t_i.

De quel ordre de grandeur peut être d_i ? Considérons que l'exécution séquentielle d'un programme est T_1 et que le temps d'exécution de l'application sur un nombre illimité de processeurs est T_∞. Dans une application parallèle on assume toujours que $T_\infty \ll T_1$. Puisque T_∞ est le chemin critique de l'application, donc pour n'importe quelle d_i on a $d_i \leq T_\infty$. En conséquence on peut supposer que d_i peut être relativement petit pour peu que le chemin critique soit petit.

Le surcoût de la reprise dépend aussi de la taille du graphe que l'application doit charger à partir du support stable et reconstruire. Cependant, on note qu'en choisissant convenablement la période τ on peut contrôler la reprise. Le choix entre un surcoût faible de sauvegarde et un surcoût plus élevé de reprise ou l'inverse devra être déterminé par l'application.

5.6 Discussion

Les types d'applications que l'on souhaite rendre tolérantes aux pannes, les types de pannes considérés et la qualité de service à assurer jouent un rôle crucial pour le choix du protocole de tolérance aux pannes. Nous concluons ce chapitre par une discussion autour des paramètres importants dans le contexte de ce travail.

Le choix du protocole de tolérance aux pannes dépend du type d'application parallèle, *i.e.* la classe du programme. Pour les applications à gros grain, le protocole SEL est plus efficace que le TIC puisque la taille du graphe de flot de données (*i.e.* le nombre de nœuds dans le graphe) représentant l'exécution d'une application à gros grain est petite (i.e. le graphe possède peu de tâches). En effet, il est plus performant de ne sauvegarder que les modifications de l'état du graphe de flot de données comme dans SEL, plutôt que de sauvegarder périodiquement l'état global du graphe comme dans TIC. En revanche, pour les applications à grain fin, le nombre de tâches devenant important, le coût de la journalisation des événements de modification dans SEL, qui est lié au nombre de tâches exécutées, devient plus important qu'une sauvegarde périodique de l'état du graphe. Ceci est particulièrement accru par le fait que dans le protocole SEL, l'enregistrement de nombreux petits messages est pénalisé par les latences réseaux. Dans le cadre du protocole TIC, le message contenant l'état d'un processus est plus important et les latences réseaux sont masquées.

L'adaptation de l'exécution tolérante aux pannes à la machine cible d'exécution et aux contraintes particulières liées à l'exécution, comme le nombre de processeurs disponibles et la durée de disponibilité de la machine, est un point critique si on ne connaît pas a priori le temps d'exécution de l'application.

La taille des données sauvegardées sur le support stable influence l'efficacité de l'exécution tolérante aux pannes. Par conséquent, la taille et le nombre de points de reprise ainsi que le choix d'un support stable centralisé ou distribué sont parmi les paramètres importants à étudier. Des expérimentations dans ce sens sont présentées dans le chapitre suivant.

Le temps de calcul perdu suite à une panne est aussi un paramètre important à étudier. La minimisation de ce temps implique un surcoût élevé car il est lié à une augmentation du nombre de points de reprise pris par processus.

Le protocole SEL fournit la possibilité de ré-exécuter une seule tâche de l'application. Ceci peut être exploité pour traiter d'autres types de pannes que les pannes franches. La ré-exécution d'une tâche sous certaines conditions peut déterminer si les résultats produits par cette tâche sont corrects ou non. Nous montrerons une utilisation du protocole SEL dans le cas du traitement des pannes par valeurs (voir §2.3).

L'étude et la compréhension de l'influence de ces paramètres et des contraintes présentées ci-dessus sur le choix du protocole de la tolérance aux pannes nécessitent des expérimentations concernant chacun de ces paramètres. Le résultat de ces expérimentations est présenté dans les chapitres suivants.

Chapitre 6

Expérimentations

Sommaire

6.1 Introduction

Ce chapitre est dédié à la validation de nos réalisations et à leur comparaison avec les travaux proches proposés dans l'environement Satin [123, 128, 127] (voir §3.9.9), sur une application simple (UB_Tree_{F*}).

Les mesures de performances présentées concernent la mise en œuvre de deux aspects importants d'un modèle de programmation parallèle et distribuée tolérant aux pannes. Ces deux aspects sont présentés comme suit :

– l'évaluation du parallélisme ;
– l'évaluation de la tolérance aux pannes.

Dans un premier temps, nous décrivons l'application choisie pour effectuer nos expérimentations. Ensuite, nous présentons les paramètres et l'environnement d'évaluation concernant KAAPI et la tolérance aux pannes. Enfin, les résultats expérimentaux obtenus avec KAAPI et leurs comparaisons avec ceux de Satin sont présentés.

Seule l'évaluation du protocole TIC est présentée dans ce chapitre ; la validation du protocole SEL et la comparaison entre les deux protocoles pour la tolérance aux pannes proposés dans ce travail (TIC et SEL) sont présentées sur une application réelle dans le chapitre suivant (chapitre 7).

6.2 Application de test : UB_Tree_{F*}

Afin de valider l'implantation de KAAPI et de son mécanisme de tolérance aux pannes, nous proposons d'étudier l'application récursive UB_Tree_{F*} : *Unbalanced Tree (Fibonacci généralisé)*. Cette application permet de comparer nos travaux avec ceux de Satin.

L'application UB_Tree_{F*} permet de calculer récursivement les éléments de la suite :

$$F(n) = F(n-1) + F(n-2) + ... + F(n-k)$$
$$F(0) = 0, F(1) = 1$$

L'application $UB\_Tree_{F*}(n, k, s)$ est une application de test dont les paramètres peuvent être modifiés pour faire varier les valeurs de travail et la longueur de chemin critique. Cette application, engendre un arbre non équilibré de profondeur maximal n ; chaque nœud non feuille engendre k nœuds fils. L'évaluation d'un nœud de valeur n se fait en parallèle si $n > s$ où s est le seuil d'arrêt de parallélisme. Sinon, l'évaluation est faite séquentiellement. La figure 6.1 illustre le graphe de flot de données associé à l'exécution de l'application UB_Tree_{F*}.

On remarque que pour n fixé, le nombre de tâches créées augmente exponentiellement lorsque s décroît. Soit $(\lambda_i)_{i=1,...,n}$ les racines du polynôme $\sum_{i=0}^{k}(-1)^i x^i$. Alors le nombre de tâches créées est asymptotiquement $O([\max(\lambda_i)]^{n-s})$.

6.3 Paramètres d'évaluation

Afin de pouvoir mesurer les performances de notre modèle de programmation parallèle distribuée tolérant aux pannes, dans cette section, nous décrivons, dans un premier temps, les paramètres d'évaluation concernant l'évaluation du parallélisme. Ensuite, les paramètres concernant l'évaluation de la tolérance aux pannes sont présentés.

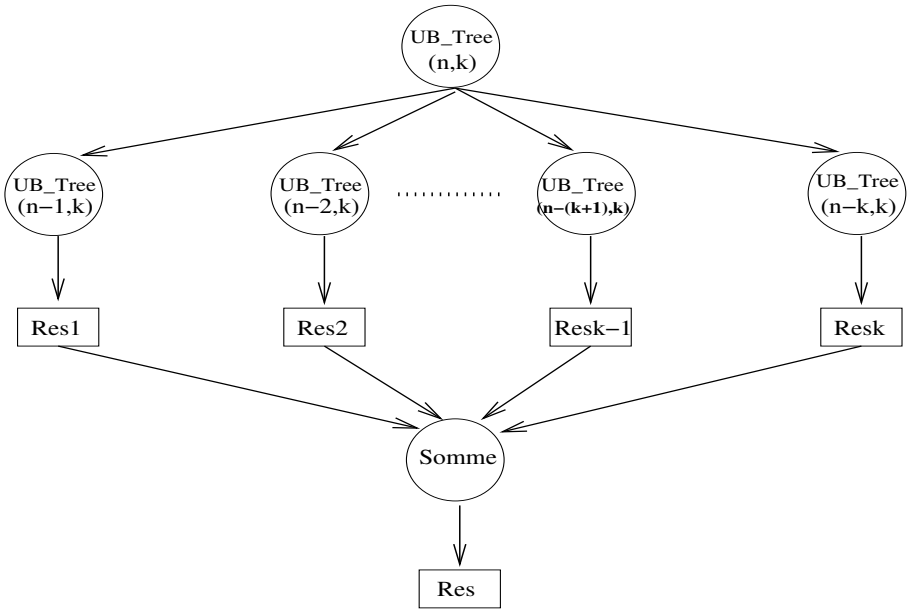

Figure 6.1 – Le graphe de flot de données représentant l'exécution de UB_Tree_{F*}.

6.3.1 Paramètres d'évaluation pour le parallélisme

En exécutant une application KAAPI et en mesurant les valeurs de certains paramètres, on peut répondre à plusieurs questions relatives aux performances :

- L'efficacité de l'implantation des primitives du langage de programmation (voir §4.5.1) ;
- L'accéleration de l'exécution du programme quand on augmente le nombre de processeurs ;
- Les communications effectuées.

Afin de répondre aux questions présentées ci-dessus, les paramètres suivants sont mesurés :

- T_s : le temps d'exécution du programme séquentiel ;
- T_1 : le temps d'exécution du programme parallèle sur un processeur ;
- T_∞ : le temps d'exécution du chemin critique du programme parallèle ;
- T_p : le temps d'exécution du programme parallèle sur p processeurs ;
- N_{theft} : le nombre de vol réussis par processeur durant l'exécution parallèle du programme ;
- σ : le nombre de tâches du programme parallèle.

6.3.2 Paramètres d'évaluation pour la tolérance aux pannes

Afin d'évaluer la tolérance aux pannes, nous considérons deux cas : une exécution sans panne et une exécution avec pannes. Une exécution normale de l'application, sans panne, permet d'évaluer le coût de la sauvegarde. Pour cela, les paramètres suivants sont considérés :

- la taille σ du graphe de flot de données répresentant l'exécution de l'application ;
- la valeur de la période τ de sauvegarde ;
- le nombre p de processeurs utilisés ;
- le nombre de supports stables (serveurs de points de reprise) utilisés durant l'exécution.

Pour l'évaluation de la reprise, les paramètres suivants sont pris en compte :
- le moment où la panne survient ;
- le nombre de pannes qui surviennent durant l'exécution de l'application.

6.4 Environnement d'évaluation

Les expérimentations que nous présentons ont été effectuées sur la grappe d'Orsay de la plateforme Grid5000. Cette grappe est constituée des éléments suivants :
- unités : 216 nœuds ;
- modèle : IBM eServer 325 ;
- cpu : AMD Opteron 246, 2 GHz / 1 MB x2 ;
- mémoire : 2 GB ;
- réseau : Gigabit ethernet (driver : tg3) ;
- disque : 80 GB / IDE (driver : amd74xx) ;
- système d'exploitation : Linux.

Dans un premier temps, nous proposons d'évaluer notre modèle de programmation parallèle, sans les méchanismes de tolérance aux pannes, et de comparer les résultats obtenus avec le modèle proposé dans Satin. Ensuite, nous évaluons le coût de la la tolérance aux pannes dans notre modèle et les comparons avec ceux de Satin.

6.5 Évaluation du parallélisme

Dans cette section, nous présentons les différentes expérimentations effectuées pour évaluer la performance de KAAPI . L'objectif principal de ces expériences est de disposer de temps de référence afin de pouvoir évaluer ce que coûtent les mécanismes de tolérance aux pannes.

Les paramètres de l'application UB_Tree_{F*} utilisés sont $n = 38$ et $k = 4$. Afin d'adapter la granularité, nous proposons, dans un premier temps, de mesurer pour cette application le tzmps d'exécution du chemin critique T_∞ qui joue un rôle important pour le protocole TIC (voir §4.4). En effet le surcoût du protocole TIC dépend du nombre de vols réussis qui est majoré par le temps d'exécution du chemin critique.

Le temps d'exécution du chemin critique : T_∞

Afin de mesurer le temps d'exécution du chemin critique pour l'application UB_Tree_{F*}, nous avons modifié le programme de cette application afin que son exécution se déroule de façon à ce que toutes les tâches du programme soient créées, mais que seules les tâches qui calculent $F(n-1)$ et les tâches qui effectuent la somme des résultats soient exécutées. Par conséquent, le temps d'exécution du programme modifié sur un processeur est le temps d'exécution du chemin critique augmenté du coût de création des tâches, noté τ_{Fork}. Alors, on a $T_\infty = (n-s)\tau_{Fork} + T_1(s)$ où $T_1(s)$ est la granularité de l'application en fonction de s.

La figure 6.2 illustre le temps d'exécution T_∞ du chemin critique, de l'application UB_Tree sur un processeur, en faisant varier le seuil d'arrêt de la découpe récursive s de 2 à 12. Chaque mesure présentée ici est la moyenne de 15 exécutions (après le filtrage des résultats et la suppression des valeurs Minimale et Maximale). Comme on peut le constater sur

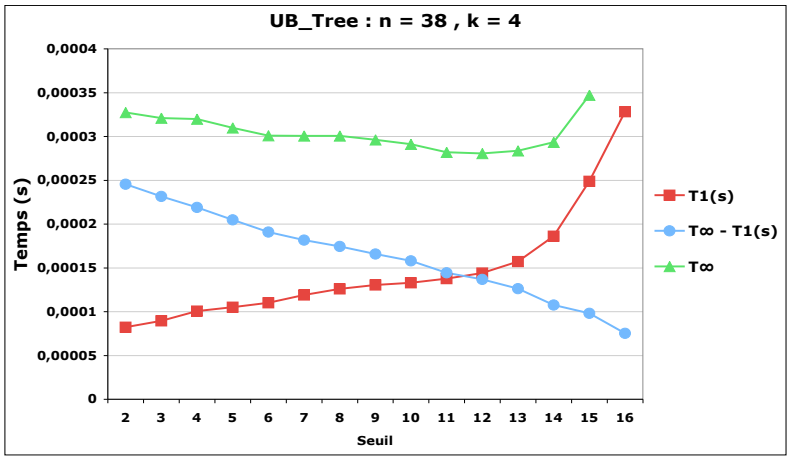

Figure 6.2 – $T_\infty(s)$ pour UB_Tree_{F*}, $n = 38, k = 4$, en faire variant le seuil s d'arrêt de parallélisme.

cette figure, le temps d'exécution du chemin critique pour cette application est minimum ($T_\infty = 0,000280582$ secondes) quand le seuil $s = 12$. On remarque que la courbe $T_\infty - T_1(s)$ décroît linéairement avec le seuil s, conformément à ce qu'on pouvait attendre.

Nombre de vols réussis

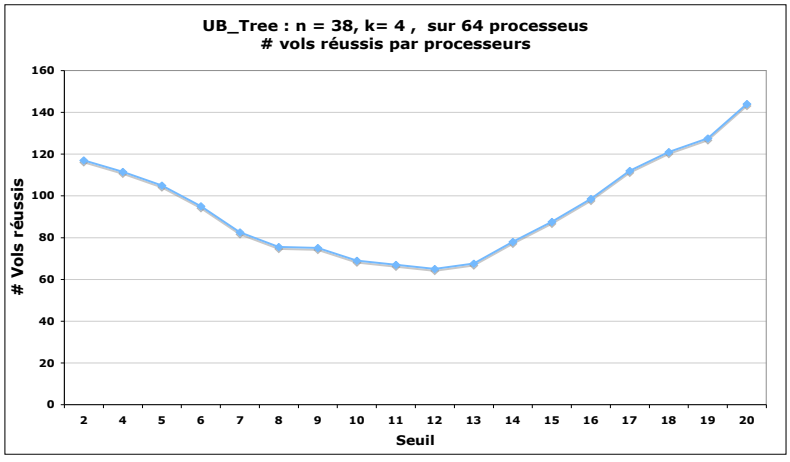

Figure 6.3 – Le nombre de vols réussis par processeur.

Comme nous l'avons présenté dans l'analyse de coût du modèle d'exécution (voir §4.4) le nombre de vols réussis par processeur dépend de T_∞ avec une grande probabilité (voir l'équation 4.3). L'objectif de cette expérience est de montrer que KAAPI vérifie cette propriété. Dans cette expérience, nous avons fixé le nombre de processeurs à 64 et lancé l'exécution du programme en faisant varier le seuil d'arrêt de la découpe récursive. Les mesures présentées dans la figure 6.3 sont les moyennes de 15 mesures effectuées après le filtrage des résultats et la suppression des valeurs Minimale et Maximale.

La figure 6.3 illustre le nombre moyen de vols réussis par processeur. Comme on peut le voir sur cette figure, le nombre minimal de vols réussis est obtenu avec un seuil de 12 ; en se basant sur les résultats de l'expérience précédente, ce seuil correspond au chemin critique $T_\infty(s)$ le plus petit. Cette expérience montre aussi que le nombre de vols réussis est très petit par rapport au nombre de tâches exécutées ; pour un seuil $s = 10$, le nombre moyen de vols réussis sur les 64 processeurs est 138 et le nombre total de tâches exécutées est 29 170 262.

Influence du grain de l'application

Cette expérience mesure l'impact du grain de l'application sur le surcoût de l'exécution de l'application pour engendrer le parallélisme ; autrement dit, cette expérience mesure l'efficacité de l'implantation des primitives du langage de programmation parallèle. Le nombre de processeurs est fixé à 1, et les mesures présentées sont les moyennes de 10 exécutions.

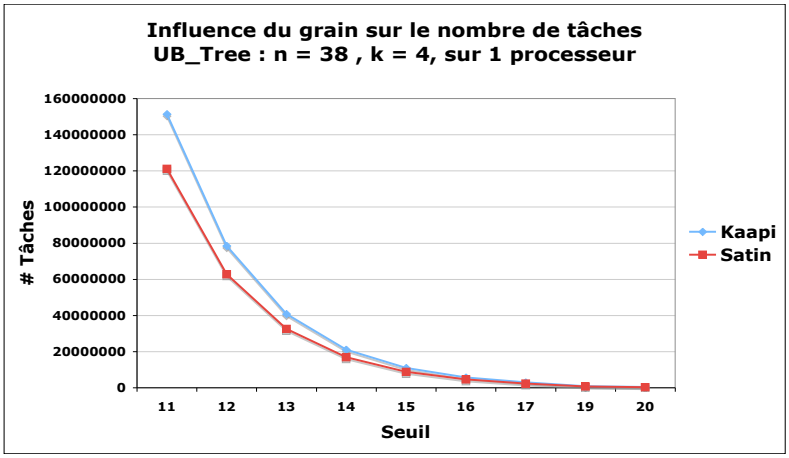

Figure 6.4 – Influence du grain de l'application sur le nombre de tâches.

La figure 6.4 représente le nombre de tâches en fonction du grain de l'application. On vérifie expérimentalement que, comme prévu par la théorie, le nombre de tâches décroît exponentiellement lorsque s croît.

Ainsi le nombre de tâches engendrées à gros grain est beaucoup moins important que celui à grain fin ; pour un seuil $s = 20$, le nombre de tâches est 329 569 pour Satin et 411 961 pour KAAPI mais pour un seuil $s = 12$, le nombre de tâches est 62 807 805 pour Satin

et 78 509 756 pour KAAPI . Dans Satin, le nombre de tâches générées est moins important que dans KAAPI ; ceci est dû au fait que, contrairement à KAAPI , dans Satin il n'y a pas de tâche engendrée pour effectuer la somme des résultats.

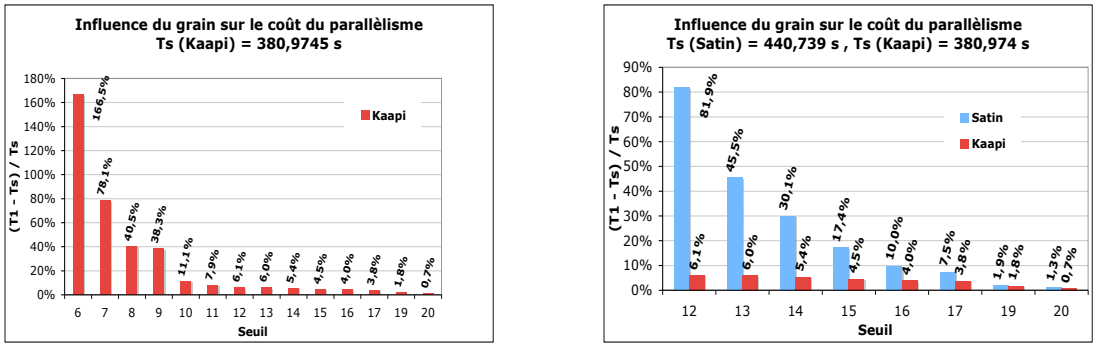

Figure 6.5 – Influence du grain de l'application sur le surcoût du parallélisme.

La figure 6.5 illustre le surcoût, par rapport à l'exécution du code séquentiel du programme, de la mise en oeuvre du parallélisme en fonction du grain de l'application. Le temps d'exécution du programme séquentiel Java pour l'application UB_Tree_{F*} : $T_s = 440,739$ secondes et celui du programme C++ est UB_Tree_{F*} : $T_s = 380,9745$ secondes.

Comme l'on peut observer sur cette figure et la figure 6.4, le surcoût de la mise en oeuvre du parallélisme dépend linéairement du nombre de tâches engendrées. Ce surcoût décroît exponentiellement lorsque le grain de l'application augmente. Les résultats obtenus montrent que le surcoût introduit par KAAPI est beaucoup moins important que celui introduit par Satin. On peut constater à l'issue de cette expérience que le surcoût introduit par KAAPI pour la mise en oeuvre du parallélisme est très faible même à grain très fin ; en effet pour un seuil $s = 6$, le programme KAAPI engendre 4026926791 tâches et le surcoût introduit ne dépasse pas 167% tandis que le programme Satin engendre 3221541433 tâches pour le même seuil et introduit un surcoût qui dépasse 4441%. Ces résultats montrent l'efficacité de l'implantation de notre modèle.

Influence du nombre de processeurs

L'objectif de cette expérience est de mesurer la variation en fonction du nombre p de processeurs du temps d'exécutions. La première constatation que l'on peut faire sur la figure 6.6 est que le temps d'exécution diminue, avec KAAPI comme avec Satin, lorsque l'on ajoute jusqu'à 64 processeurs. Sur 128 processeurs, les temps d'exécution KAAPI et Satin restent équivalent à ceux sur 64 processeurs ; ceci s'explique par le fait que l'algorithme d'ordonnancement de type vol de travail utilisé dans KAAPI et Satin est probabiliste (le choix d'un processus victime est fait de façon probabiliste) et ne fonctionne pas bien pour l'application UB_Tree_{F*} avec les paramètres considérés.

Afin de montrer le surcoût introduit par l'augmentation du nombre de processeurs, nous illustrons dans la figure 6.7, le surcoût introduit par rapport à T_1. Le calcul de $\frac{p*T_p - T_1}{T_1}$ montre

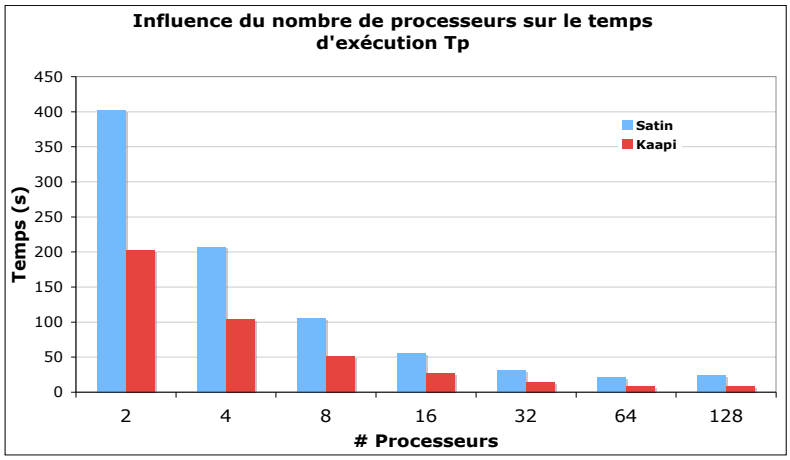

Figure 6.6 – Influence du nombre de processeurs.

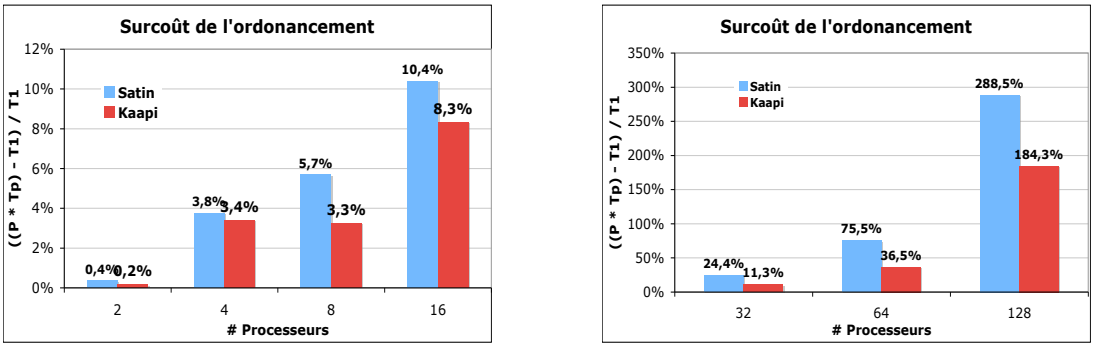

Figure 6.7 – Influence du nombre de processeurs sur le surcoût de l'ordonancement.

le surcoût de l'algorithme d'ordonnancement (vol de travail) utilisé dans KAAPI et Satin. Le surcoût par rapport à T_s est montré dans la figure 6.8. En effet le calcul de $\frac{p*T_p - T_s}{T_s}$ montre le surcoût total du parallélisme introduit. On remarque que le surcoût de KAAPI augmente linéairement avec le nombre de processeurs p, ce qui est cohérent avec la borne :

$$\frac{pT_p - T_s}{T_s} = \frac{p(\frac{T_1}{p} + O(T_\infty)) - T_s}{T_s} = \frac{T_1}{T_s} - 1 + O(p\frac{T_\infty}{T_s}) = O(1) + O(p\frac{T_\infty}{T_s})$$

On peut conclure de cette expérience, des résultats obtenus avec KAAPI et de leur comparaison avec ceux obtenus avec Satin que l'efficacité de l'implantation de notre modèle de programmation et d'exécution est démontrée.

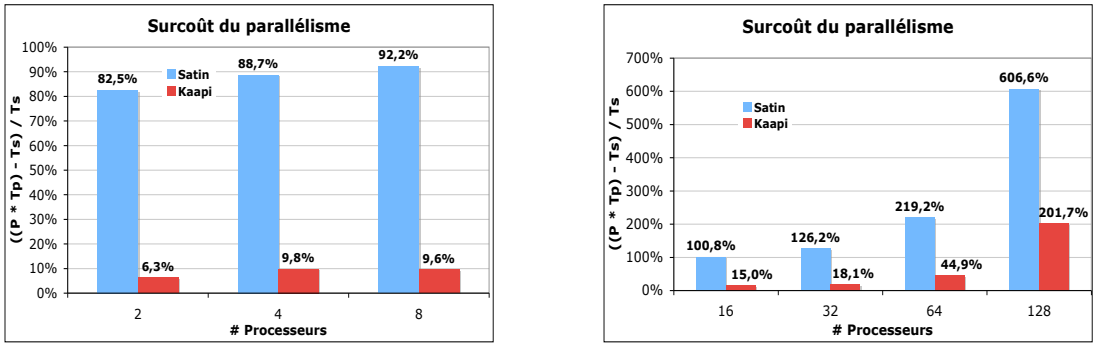

Figure 6.8 – Influence du nombre de processeurs sur le surcoût du parallélisme par rapport à une exécution séquentielle.

6.6 Évaluation de la tolérance aux pannes

Nous proposons dans cette section d'évaluer la performance du protocole TIC implanté dans KAAPI et de comparer les résultats obtenus avec le protocole de tolérance aux pannes proposé dans Satin. Nous commençons cette section par l'évaluation de la sauvegarde, ensuite nous évaluons la reprise. Nous notons que chaque mesure présentée dans cette section est la moyenne de 15 expériences (après le filtrage des résultats et la suppression des valeurs Minimal et Maximal) et la taille moyenne d'un point de reprise est 4 Ko.

6.6.1 Sauvegarde

Comme nous l'avons vu dans l'analyse théorique (§5.5), le protocole TIC dépend de plusieurs paramètres : la valeur de la période de sauvegarde, le nombre de vols réussis et le temps d'accès au support stable (serveur de points de reprise). Afin d'évaluer les performances de la sauvegarde dans le protocole TIC, nous proposons de mesurer l'influence de chacun de ces paramètres dans une exécution sans panne.

6.6.1.1 Influence de la période de sauvegarde

Dans cette expérience, nous avons fixé le nombre de processeurs à $p = 8$ et nous avons utilisé un support stable de stockage (serveur centralisé). Les paramètres de l'application de test sont $n = 38$, $k = 4$ et $s = 12$. La période de sauvegarde varie de 100 à 10 ms.

Comme l'illustre la figure 6.9, le temps d'exécution total est intimement lié à la valeur de la période de sauvegarde : lorsque la période augmente d'un facteur k le temps d'exécution diminue d'un facteur k. Ceci s'explique par le fait que le nombre de sauvegardes devient non négligeable lorsque la période diminue : dans notre cas et pour cette application de 100 secondes, une période de 50 ms va entraîner 2000 requêtes de sauvegarde locale par processus (soit 16 000 requêtes au total). Ce flux supplémentaire engendre des problèmes de congestion aussi bien sur l'interface réseau du support de stockage que sur les unités de stockage

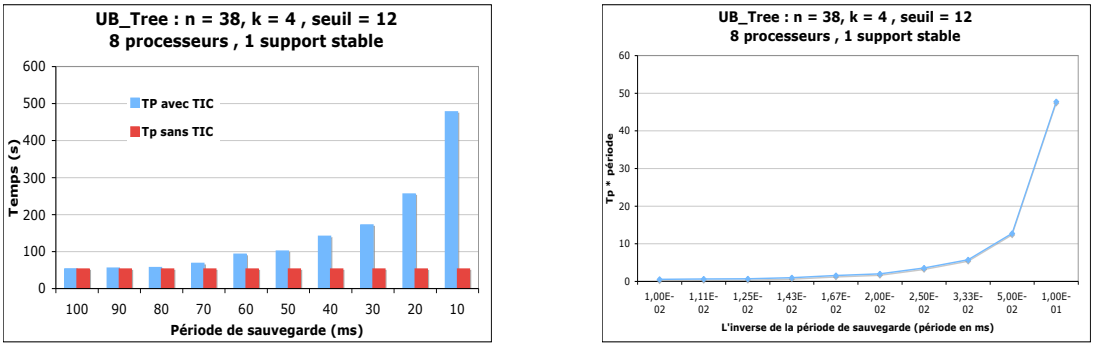

Figure 6.9 – Influence de la période de sauvegarde.

sous jacentes. L'augmentation du nombre de supports de stockage durant l'exécution est une solution pour réduire le temps d'exécution.

Influence du nombre de supports stables

En augmentant le nombre de supports de stockage (serveurs de points de reprise) utilisés par l'application, le temps d'exécution, avec une période de sauvegarde "petite", doit diminuer. La figure 6.10 montre le temps d'exécution, sur 8 processeurs, avec une période de 50 ms en fonction du nombre de supports de stockage utilisés. En utilisant deux supports de stockage, le temps d'exécution passe de 100 à 53 secondes. La figure 6.11 illustre que pour

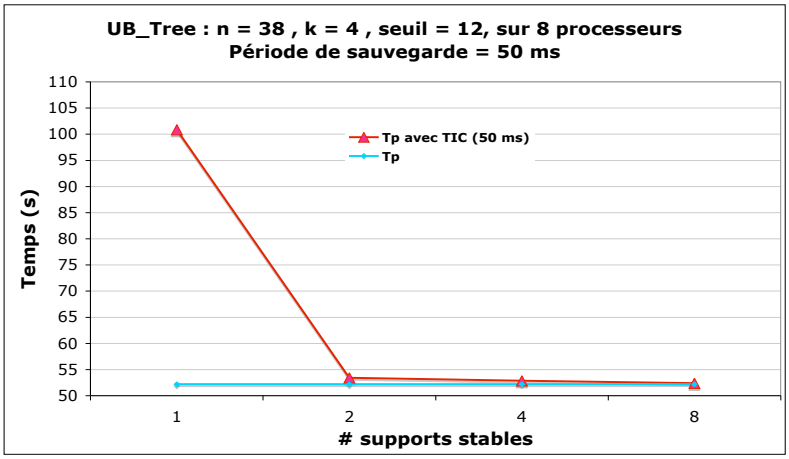

Figure 6.10 – Influence du nombre de supports stables sur 8 processeurs.

une exécution de l'application sur 128 processeurs, avec une période de 50 ms, l'utilisation

de 16 supports de stockage masque le surcoût de la sauvegarde sur l'exécution.

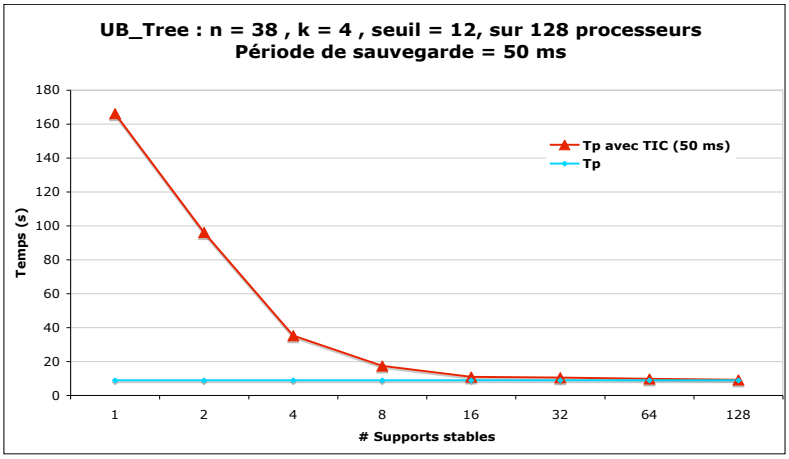

Figure 6.11 – Influence du nombre de supports stables sur 128 processeurs.

6.6.1.2 Influence du vol de travail

Le second paramètre important pour le protocole TIC, comme présenté dans §5.5 et §4.4, est le nombre de vols réussis durant l'exécution. Ce nombre dépend d'une part du temps d'exécution du chemin critique T_∞ et d'autre part, du nombre de processeurs à l'exécution. Afin d'étudier l'influence du vol de travail sur le protocole TIC, nous proposons d'évaluer son coût d'abord en fonction de T_∞ et ensuite en fonction du nombre de processeurs.

Influence de T_∞

Afin de mesurer l'impact de T_∞ sur l'exécution avec TIC, dans cette expérience, on exécute notre application, avec les paramètres n = 38 et k = 4 sur 16 processeurs, avec une valeur de période de sauvegarde locale infinie (pour en supprimer l'influence) et en utilisant 16 supports de stockage. Le choix de la période à $+\infty$ secondes est fait pour éliminer la sauvegarde locale de points de reprise et de ne mesurer que l'impact de la sauvegarde des points de reprise forcés. Le choix de 16 supports de stockage quand à lui est fait pour éliminer tout problème de congestion de ceux-ci.

La figure 6.12, montre le surcoût du TIC par rapport au temps d'exécution sans TIC en fonction de T_∞. Comme on peut le constater sur cette figure, le surcoût du TIC dépend du chemin critique de l'application.

Influence du nombre de processeurs

Dans cette expérience, nous faisons varier le nombre de processeurs de notre application, avec les paramètres $n = 38, k = 4, s = 12$ et 16 supports de stockage. La figure 6.13 illustre

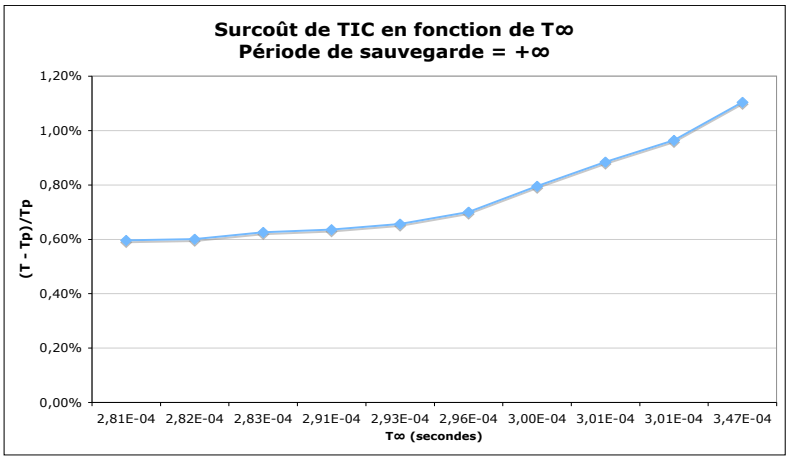

Figure 6.12 – Influence du chemin critique (T_∞) sur l'exécution avec TIC.

le temps d'exécution T_p, sans TIC, et le temps d'exécution en utilisant TIC avec deux périodes : $\tau = +\infty$ et τ = 1 seconde. Comme on peut le constater, le temps d'exécution avec TIC, comme sans TIC, diminue lorsque le nombre de processeurs augmente et le surcoût introduit par le protocole TIC est faible.

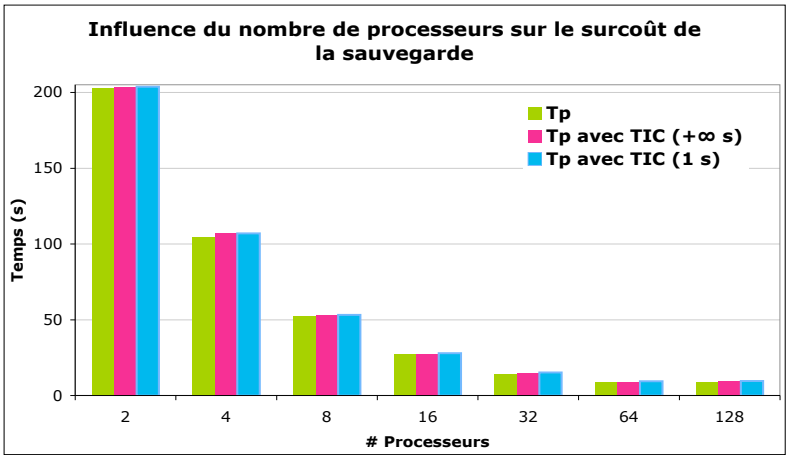

Figure 6.13 – Influence du nombre de processeurs sur l'exécution avec sauvegarde.

Cette expérience permet de montrer le surcoût réel introduit par notre protocole lorsqu'augmente le nombre de processeurs et donc le nombre de vols. La figure 6.14 illustre le

surcoût de TIC, avec une période à $+\infty$ seconde, par rapport à l'exécution parallèle sans le protocole. Cette figure montre que le surcoût introduit par le protocole augmente de façon linéaire en fonction du nombre de processeurs ; le surcoût sur 128 processeurs dans notre cas, ne dépasse pas 6% du temps d'exécution sans le protocole TIC.

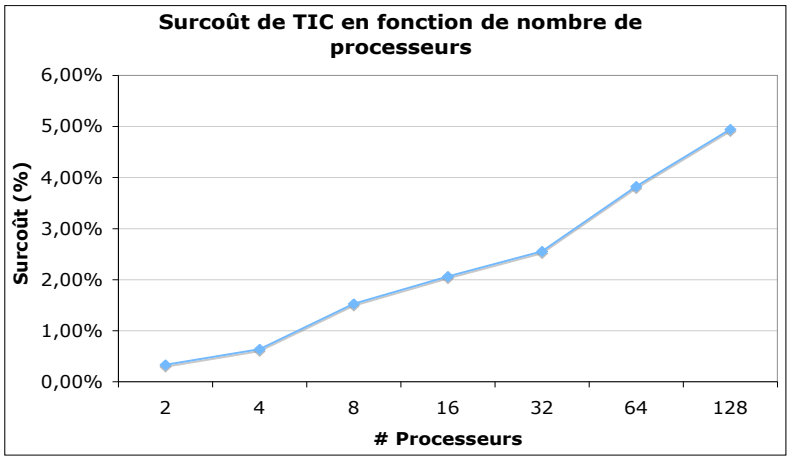

Figure 6.14 – Influence du nombre de processeurs sur l'exécution avec sauvegarde.

6.6.2 Reprise

Nous proposons dans cette section d'évaluer l'impact de la présence de pannes sur l'exécution de notre application de test. Dans l'expérience réalisée, la panne franche d'un processus est produite par l'envoi d'un signal *kill* à celui-ci. Les paramètres de notre application sont $n = 38, k = 4, s = 12$ et un seul support de stockage. Un processus défaillant est remplacé par un nouveau processus.

La figure 6.15 montre le temps d'exécution, sur 8 processeurs, en fonction du nombre de pannes. Comme on peut l'observer sur cette figure, le temps d'exécution augmente de façon linéaire avec le nombre de pannes.

La figure 6.16 illustre le surcoût relatif de l'expérience précédente. En effet le calcul de $\frac{T_f - T_{chk}}{T_{chk}}$, où T_f est le temps d'exécution avec pannes et T_{chk} le temps d'exécution avec TIC sans panne, montre le coût de la reprise en fonction du nombre de pannes. Ainsi, le calcul de $\frac{T_f - T_p}{T_p}$ montre le surcoût total de TIC en fonction du nombre de pannes. On remarque que le temps d'exécution augmente linéairement avec le nombre de pannes.

6.6.3 Comparaison avec Satin

Dans cette section, nous comparons notre approche pour la tolérance aux pannes avec celle de Satin. Nous avons choisi la comparaison avec cet environnement car le modèle de programmation parallèle de Satin est proche de celui de KAAPI . En effet le modèle

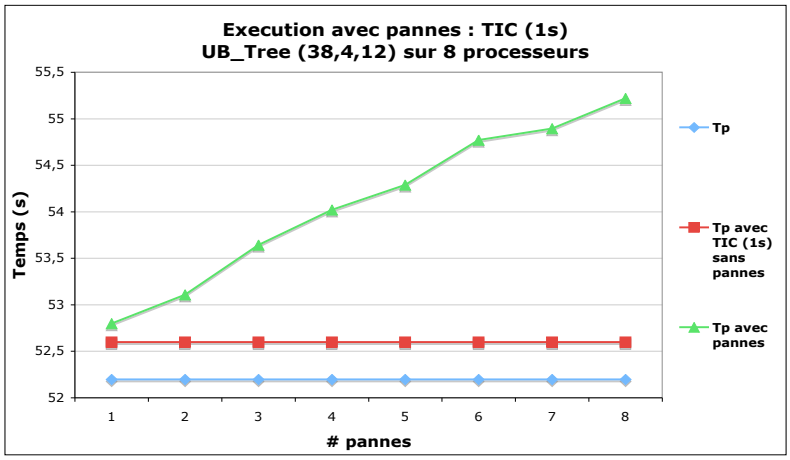

Figure 6.15 – Influence du nombre de pannes sur le temps d'exécution

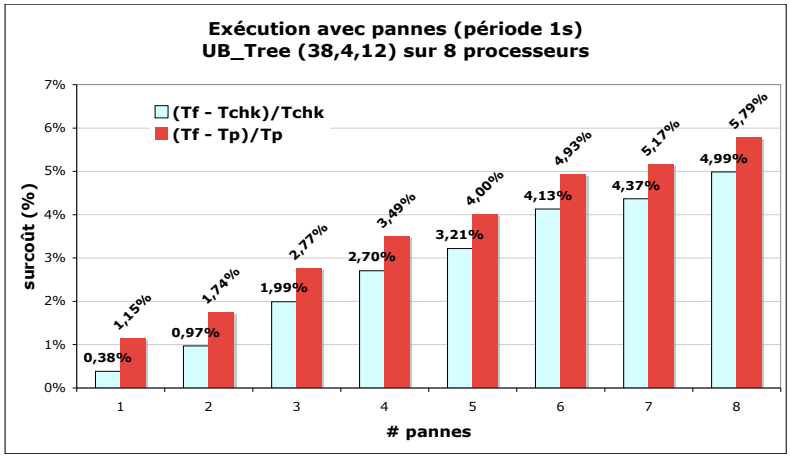

Figure 6.16 – Surcoût de TIC en fonction de nombre de pannes.

de programmation de Satin (série parallèle) est un cas spécial de celui de KAAPI (flot de données) et Satin utilise aussi un ordonnancement par vol de travail aléatoire.

Dans cette expérience, nous évaluons l'impact du moment de panne sur l'exécution par rapport à une exécution sans protocole de tolérance aux pannes. L'application de test UB_{Tree} avec les paramètres $n = 38, k = 4, s = 12$ est utilisée dans cette expérience avec un seul support de stockage et une période de sauvegarde $\tau = 1$ seconde utilisée pour l'expérience avec KAAPI . Une panne franche est produite durant l'exécution par l'envoi d'un signal *kill*

à un processus de l'exécution.

La figure 6.17 montre les résultats des expériences suivantes obtenues avec Satin (resp. KAAPI) sur 8, 16 et 32 processeurs :

1. une exécution avec le protocole de tolérance aux pannes mais sans panne ;
2. une exécution avec le protocole de tolérance aux pannes et une panne après une durée égale à 25% du temps de l'exécution ;
3. une exécution avec le protocole de tolérance aux pannes et une panne après une durée égale à 50% du temps de l'exécution ;
4. une exécution avec le protocole de tolérance aux pannes et une panne après une durée égale à 75% du temps de l'exécution.

Le surcoût relatif à l'exécution de l'application sans protocole de tolérance aux pannes, des quatre cas précédents, est illustré dans la figure 6.17. Comme on peut le constater sur cette figure, le surcoût introduit par le protocole TIC (KAAPI) est très faible et ne dépend pas du moment de la panne. En revanche pour Satin, le surcoût dépend du moment de la panne ; plus la panne apparaît tard plus le surcoût est important. Ceci est dû au protocole de tolérance aux pannes, dans Satin on ne sauvegarde les résultats des tâches orphelines qu'après la détection de pannes (voir §3.9.9).

6.7 Conclusion

Ce chapitre, a décrit différentes expériences permettant de valider l'implantation de KAAPI et du protocole TIC pour la tolérance aux pannes. Ces expériences sont divisées en deux catégories d'évaluation :

– l'évaluation du support d'exécution (KAAPI) permettant d'obtenir des mesures de référence afin d'évaluer le protocole de tolérance aux pannes ;
– l'évaluation du protocole TIC permettant de valider son implantation et ses propriétés.

L'application de test considérée dans ce chapitre est une application simple dont le code ne dépasse pas quelques dizaines de lignes. Le chapitre suivant est consacré à des expérimentations effectuées sur une application beaucoup plus importante, l'application QAP. Ceci permet de valider KAAPI et ses protocoles de tolérance aux pannes dans un contexte beaucoup plus réaliste.

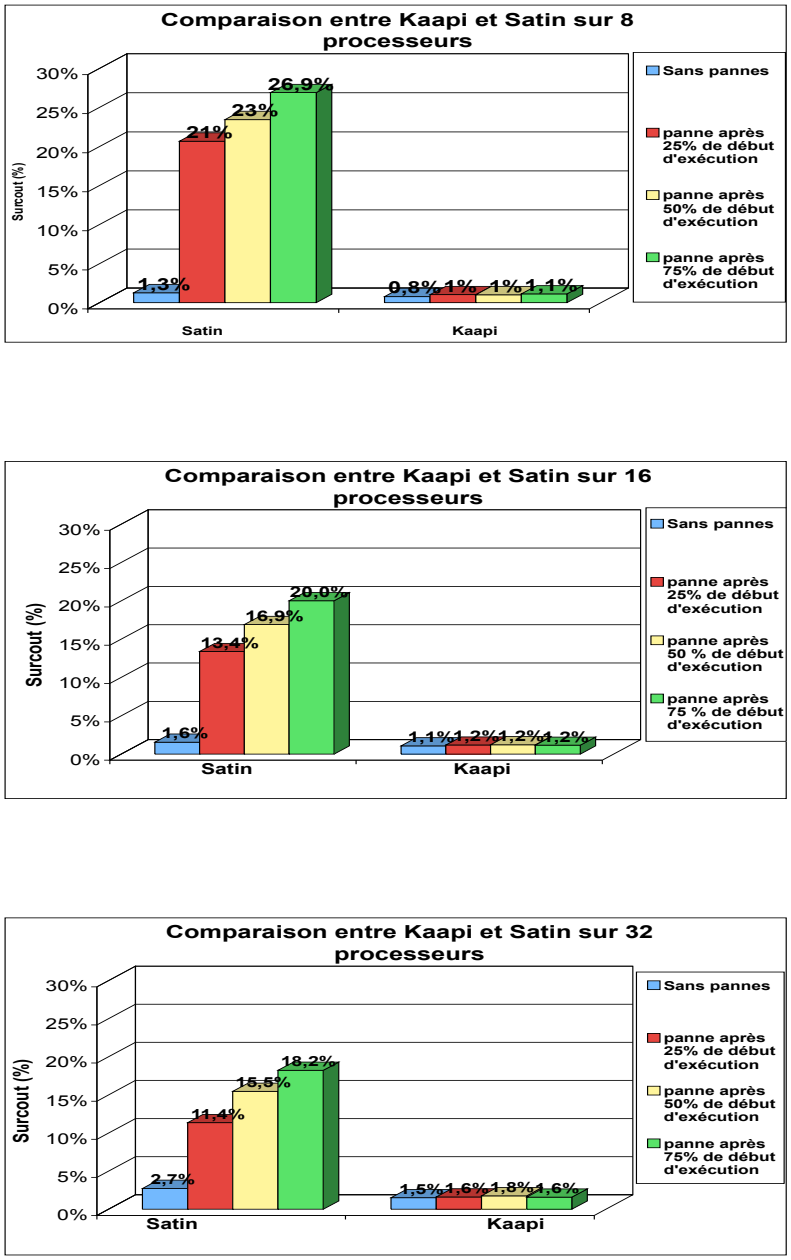

Figure 6.17 – Comparaison du surcoût de la tolérance aux pannes entre KAAPI et Satin sur 8, 16 et 32 processeurs.

Chapitre 7

Expérimentations avec l'application QAP

Sommaire

7.1 Introduction

Ce chapitre est consacré à la description des travaux menés dans le cadre de l'ACI DOC-G (Défi en Optimisation Combinatoire sur Grille) et du projet AHA (Algorithmes Hybrides et Adaptatifs). Ces travaux ont pour objectif d'exploiter des architectures de type grille de calcul pour la résolution de problèmes d'optimisation combinatoire de type Branch-and-X (Branch-and-Bound, Branch-and-Cut ou Branch-and-Price). Nous proposons donc le portage de l'application QAP (Quadratic Assignment Problem) sur notre support tolérant les pannes afin de permettre l'exploitation fiable des architectures de grande taille (grappe ou grille). Pour cela, nous avons utilisé une version de l'application QAP développée sur la bibliothèque Bob++ développée au laboratoire PRISM à Versailles. L'objectif de Bob++ est de faciliter le développement de telles applications. Dans la suite de ce chapitre, nous présentons d'abord le problème QAP. Puis, nous décrivons les développements effectués, enfin nous présentons les différentes expérimentations réalisées et les résultats qui en découlent.

7.2 Présentation du problème

Le Problème d'Affectation Quadratique (voir figure 7.1) est l'un des problèmes d'Optimisation Combinatoire les plus connus. Identifié en 1957 par Koopman et Beckmann [69], il consiste à trouver le placement à moindre coût de n unités communiquant entres elles à partir de n emplacements prédéterminés. Dans la théorie de localisation, le problème est représenté par deux matrices des flux f et des distances d. Dans la très grande majorité des applications du problème, le coût c_{ijkl} d'interaction entre l'unité i placée sur le site j et l'unité k placée sur le site l est considéré comme proportionnel au produit du flux échangé entre les deux unités (f_{ik}) par la distance entre les deux sites (d_{jl}). Ce problème d'affectation quadratique appartient à la classe des problèmes *NP-difficiles*.

Le problème s'écrit donc :

$$(QAP1) : Min \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} f_{ij}d_{kl}x_{ij}x_{kl}$$

$$x_{ij} = 0,1 \qquad i,j = 1,.....,n \tag{7.1}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1,.....,n \tag{7.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1,.....,n \tag{7.3}$$

La simplicité de la formulation de ce problème est certainement une des raisons de son succès. Par ailleurs, le très grand nombre d'applications concrètes de ce problème dans des domaines très variés (Informatique, Electronique, Architecture, conception d'ateliers,...) engendre une forte demande de méthodes de résolution efficaces ce qui a, également, concouru à ce succès. Enfin, la très grande difficulté de ce problème permet de comprendre la continuité de l'intérêt qui y est porté.

Une instance du QAP est caractérisée par sa taille n et trois matrices carrées de réels. Lorsque le QAP est considéré comme un problème de placement de n éléments sur n emplacements, ces matrices sont désignées comme suit :

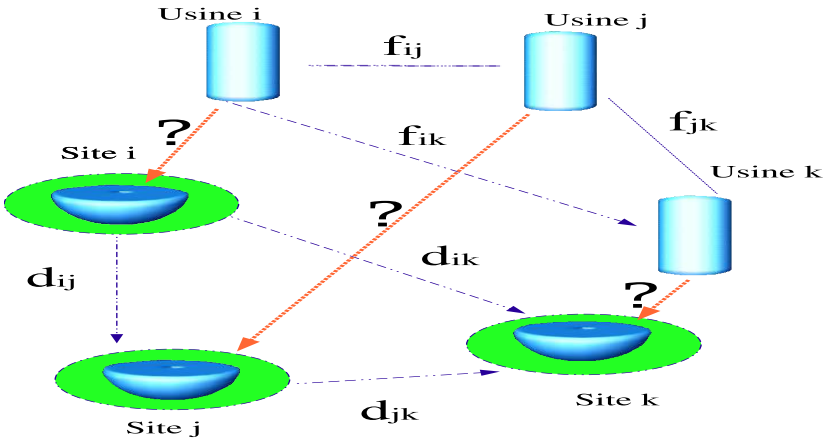

Figure 7.1 – Porblème d'Affectation Quadratique [41]

- (D_n) : la matrice des distances entre les emplacements
- (F_n) : matrice des flux entre les éléments
- (C_n) la matrice des coûts fixes des affectations des éléments sur les emplacements.

Communément, on distingue deux types d'approches dans les méthodes de résolution pour le QAP : les méthodes exactes et les méthodes suboptimales. Les méthodes exactes résolvent, au sens propre le QAP. Autrement dit, elles fournissent , à coup sûr, un optimal global de l'instance. Cependant, les méthodes exactes conçues à ce jour ont un coût de calcul qui croît très vite avec la taille de l'instance à résoudre.

Les algorithmes Branch-and-Bound (B&B) sont les méthodes exactes les plus efficaces pour le QAP [88]. Les méthodes B&B reposent sur l'exploration intélligente d'une arborescence de construction des solutions. Au sommet de l'arbre, on place une solution partielle vide ; puis pour passer d'un niveau de l'arbre au suivant, on développe les différentes affectations possibles sur un site non affecté en amont. L'ensemble des feuilles de l'arbre est constitué de toutes les solutions complètes ; c'est l'espace de recherche de l'instance à résoudre. Afin de limiter l'exploration de l'arbre, les méthodes B&B calculent une borne inférieure des sous-arbres. Plus la borne est précise et rapide à calculer, plus l'exploration est réduite.

Un algorithme B&B est caractérisé par les éléments suivants : borne inférieure, borne supérieure, strategie de parcours (sélection ou recherche) et stratégie de branchement.

7.3 Développements effectués pour le QAP

Comme nous venons de voir, le problème du QAP consiste à affecter n unités communicant entre elles (usines) à n emplacements disponibles de manière à réduire le coût d'exploitation de l'ensemble des usines.

Un code de cette application écrit en Bob++ et parallélisé en utilisant l'interface ATHAPASCAN était disponible au début de ces travaux [101]. Ce code utilise un algorithme de type

Branch-and-Bound proposé par Bob++. La parallélisation proposée dans ce code consiste à paralléliser la bibliothèque Bob++ par l'utilisation d'ATHAPASCAN en décomposant l'arborescence parcourue lors de la recherche dans l'espace de solutions. Le code initial de l'application QAP n'est donc pas modifié et ne contient aucun appel aux primitives ATHAPASCAN. Les principaux travaux de développement ont consisté au portage du code dans KAAPI , à l'intégration de la tolérance aux pannes et à l'optimisation du code.

Afin de prendre en compte des données de l'application qui ne font pas partie de l'état à sauvegarder, comme par exemple des variables globales, deux méthodes peuvent être fournies par l'application : la première, *application_store*, est appelée lors d'une sauvegarde et permet à l'application de définir ses propres données à sauvegarder ; la seconde, *application_load*, est appelée lors d'une restauration et permet à l'application de restaurer ses variables globales. Dans le cas où, un état d'exécution est restauré dans le contexte d'un processus s'exécutant, l'application doit gérer, dans la définition de *application_load*, la cohérence des variables globales.

7.4 Expérimentations sur grappe

Les objectifs de ces expériences sont de valider et d'illustrer les modèles de coût algorithmique de nos protocoles de tolérance aux pannes présentés dans la section 5.5 en utilisant une application réelle et un nombre important de processeurs. Les expériences ont été effectuées sur le iCluster2[1] de l'IMAG hébergé à l'INRIA Rhône-Alpes en utilisant plusieurs instances, «NUGENT[2]», du QAP. Cette grappe est composé de 104 biprocesseurs Itanium-2 (900 MHz) interconnectés par un réseau Fast Ethernet 100 Mo/s.

7.4.1 Influence du grain de l'application sur 1 processeur

Cette expérience mesure l'impact du grain de l'application sur le surcoût de l'exécution de l'application. Le nombre de processeurs est fixé à 1 et le nombre de supports de stockage est aussi fixé à 1.

La figure 7.2 montre le nombre de tâches créées par l'application QAP en fonction du seuil d'arrêt de création du parallélisme pour l'instance NUGENT17S. Pour cette instance, à partir du seuil 6 le nombre de tâches engendrées par l'application est constant (7491).

La figure 7.3 compare les temps d'exécution avec et sans le protocole SEL en fonction du seuil d'arrêt de création du parallélisme de l'application. La courbe «théorique[3]» représente le temps donné par notre modèle. Pour un petit grain (seuil = 5, 6 ou 10), le nombre de tâches est important et le surcoût dû au protocole représente plus de 40% du temps d'exécution. En augmentant le volume de calcul en utilisant l'instance NUGENT18S du QAP, ce surcoût est d'environ 10% pour environ 4000 tâches pour un temps d'exécution de $2286s$.

La figure 7.4 compare les temps d'exécution avec et sans le protocole TIC, avec une période de sauvegarde $\tau = 20$ secondes, en fonction du seuil d'arrêt de création du parallélisme

[1] http ://i-cluster2.inrialpes.fr

[2] http ://www.seas.upenn.edu/qaplib/inst.html#NVR.

[3] La courbe théorique a été obtenue, à partir d'une analyse spécialisée de coût présentée dans l'annexe A, après estimation des paramètres ($t_s \simeq 7.16E^{-5}$, $\gamma^* \simeq 1.009 + \sigma t_s$).

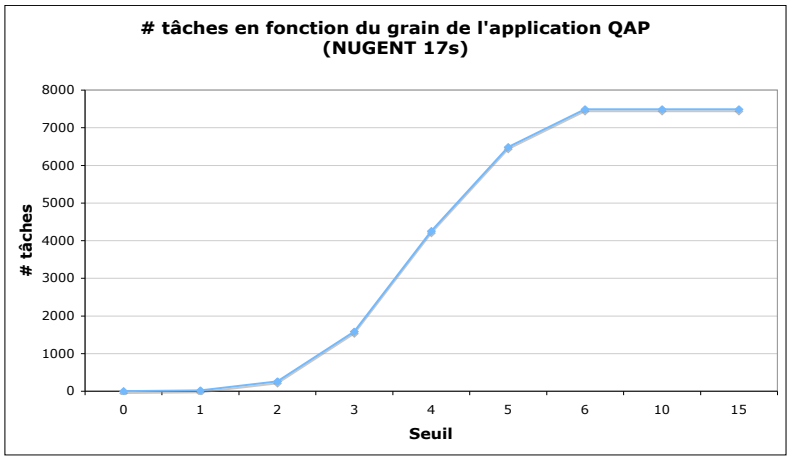

Figure 7.2 – Nombre de tâches engendrées par l'application pour NUGENT17S en fonction du seuil d'arrêt de parallélisme

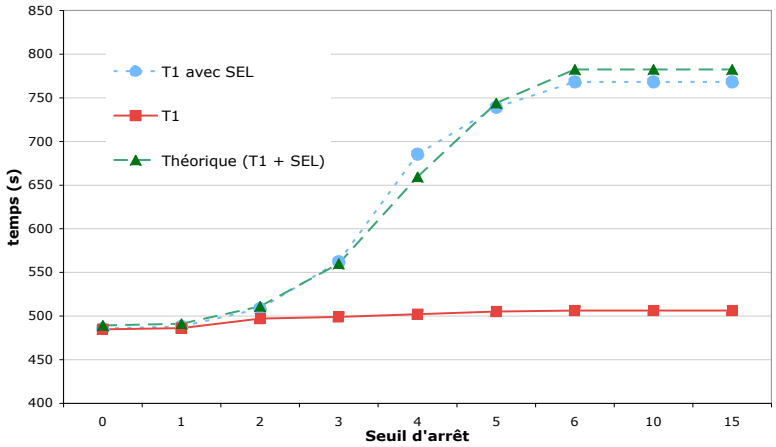

Figure 7.3 – NUGENT17S sur 1 processeur : Temps d'exécution, avec le protocole SEL, en fonction du seuil d'arrêt.

de l'application. La courbe «théorique[4]» représente le temps donné par notre modèle. Il faut noter que, ici, il n'y a pas de points de reprise forcés car le nombre de processeurs est fixé à 1 dans cette expérience. Les résultats des figures 7.3 et 7.4 montrent la pertinence de notre

[4]La courbe théorique a été obtenue, à partir d'une analyse spécialisée de coût présentée dans l'annexe A, après estimation des paramètres ($\tilde{t_s} \simeq 8,14E^{-5}$, $\gamma^\sharp \simeq 1.002 + N\tilde{t_s}$).

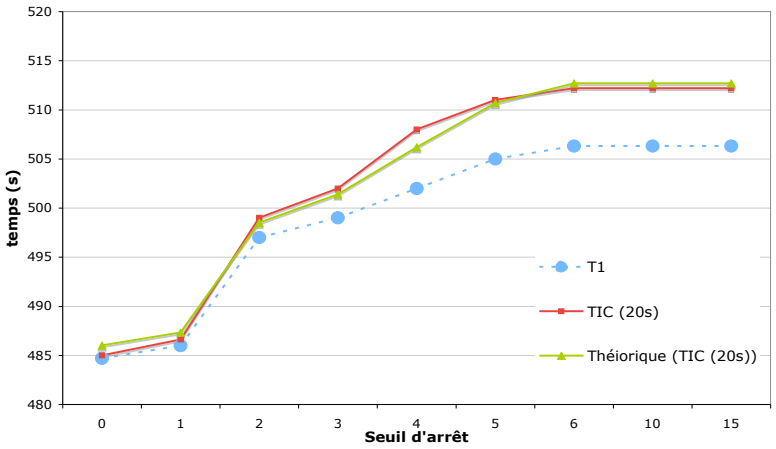

Figure 7.4 – NUGENT17S sur 1 processeur : Temps d'exécution, avec le protocole TIC, en fonction du seuil d'arrêt.

modèle.

La figure 7.5 montre les différences de temps entre le protocole SEL et le protocole TIC pour deux périodes (une sauvegarde toute les secondes et toute les 20 secondes). Les mesures de la méthode TIC montre que le surcoût est très faible vis-à-vis de l'exécution sans sauvegarde (entre 0.6% et 1%).

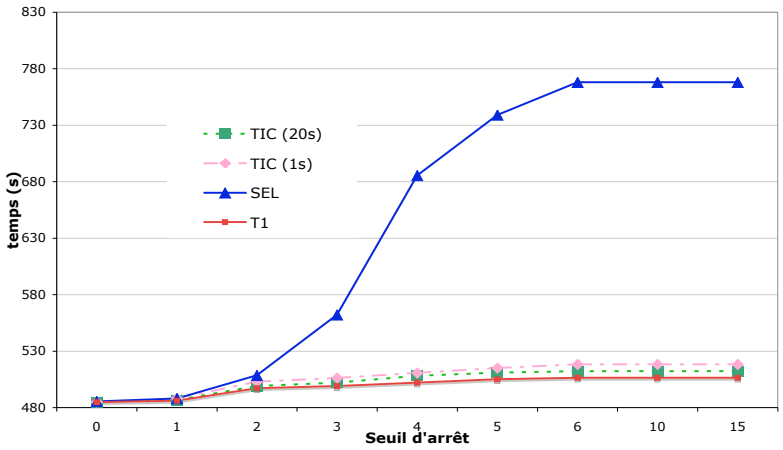

Figure 7.5 – NUGENT17S sur 1 processeur : Comparaison des protocoles SEL et TIC.

L'écart entre les deux protocoles montre un surcoût important de l'approche SEL dû à la

sauvegarde d'un grand ensemble d'événements. Réduire ce coût demande soit d'augmenter le grain de l'application et donc de perdre en degré de parallélisme et en efficacité ; soit de retarder les écritures sur le support stable en désynchronisant la sauvegarde effectives des événements, et donc en utilisant une approche de type TIC.

7.4.2 Influence du grain de l'application sur 120 processeurs

Cette expérience mesure l'impact du grain de l'application sur le surcoût du temps d'exécution de l'application. Le nombre de processeurs est fixé à 120. Afin de mesurer le surcoût introduit par les deux mécanismes (TIC et SEL) et non pas le surcoût associé à l'utilisation d'un support de stockage centralisé, le nombre de supports de stockage est également fixé à 120.

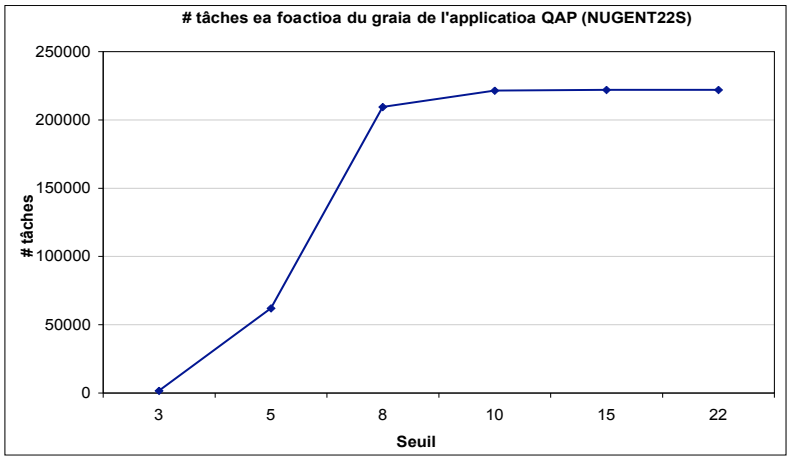

Figure 7.6 – Nombre de tâches générées par l'application pour NUGENT22S en fonction du seuil d'arrêt de parallélisme

La figure 7.6 montre le nombre de tâches créées par l'application QAP en fonction du seuil d'arrêt de création du parallélisme pour l'instance NUGENT22S. Pour cette instance, à partir du seuil 10 le nombre de tâches engendrées par l'application est constant (221480).

Le temps d'exécution séquentielle de cette instance sans KAAPI est $T_s = 34695$ secondes. Avec KAAPI et à grain fin (seuil ≥ 10), l'exécution sur un seul processeur génère 221480 tâches et elle dure $T_1 = 34845$ secondes.

L'impact du grain de l'application sur le surcoût de l'exécution de l'application est illustré dans la figure 7.7. Le nombre de tâches engendrées a une influence directe sur le coût des protocoles. Comme on le constate sur cette figure, le surcoût de SEL est très sensible au nombre total de tâches comme prévu par l'équation 5.3. Cette figure montre aussi l'impact du grain de l'application sur le surcoût de TIC deux périodes ($\tau = 1$ et $\tau = 20$ secondes). Comme le montre l'inégalité 5.5 et qu'on le constate sur cette figure, le surcoût dépend du chemin critique et de la période τ. Les résultats de la figure 7.7 confirme ce que prévoit la théorie.

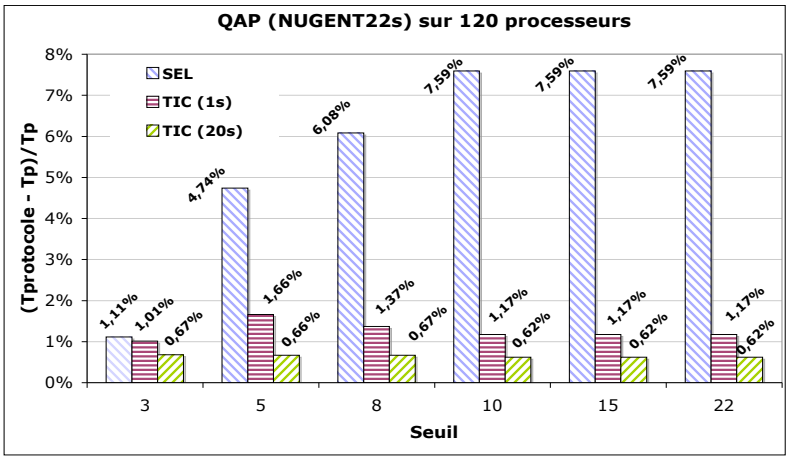

Figure 7.7 – L'impact du grain de l'application QAP (NUGENT22S) sur 120 processeurs.

7.4.3 Influence du nombre de processeurs

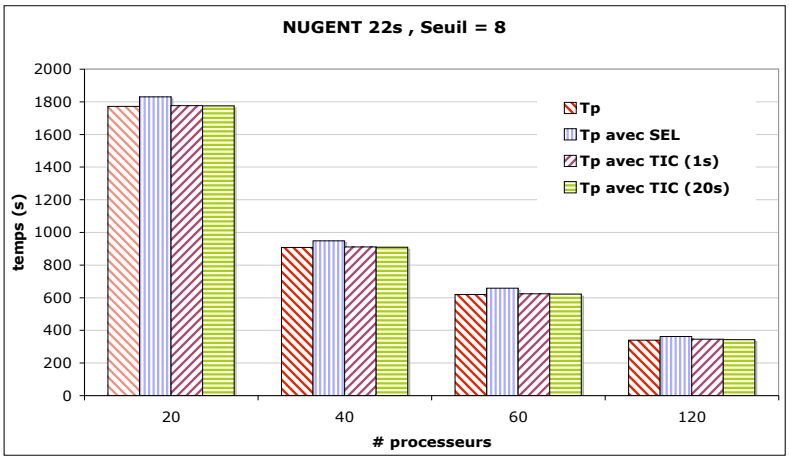

Figure 7.8 – NUGENT22S : Temps d'exécution en fonction de nombre de processeurs.

La figure 7.8 montre le gain en fonction du nombre p de processeurs des exécutions sur l'instance NUGENT22S du QAP (sans et avec les protocoles de tolérance aux pannes). Dans cette expérience, le nombre de supports de stockage est égal au nombre de processeurs utilisés pour l'application et le seuil d'arrêt de la création du parallélisme est fixé à 8. On remarque sur cette figure que le temps d'exécution avec les deux protocoles, comme sans

protocole de tolérance aux pannes, diminue lorsque l'on ajoute des processeurs.

7.4.4 Taille des points de reprise

Dans cette section, nous nous intéressons à la taille des points de reprise sauvegardés sur le support de stockage en utilisant les deux protocoles TIC et SEL.

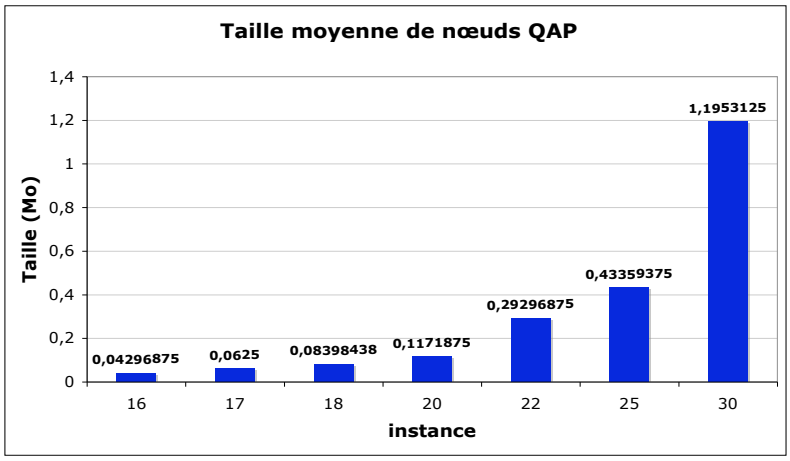

Figure 7.9 – Taille moyenne de nœuds de l'application en fonction de l'instance QAP.

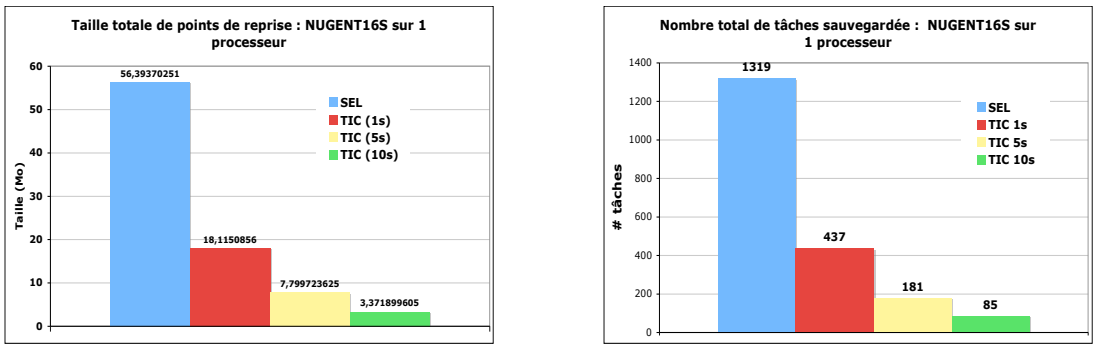

Figure 7.10 – NUGENT16S sur 1 processeur : Taille totale de points de reprise sauvegardés sur le support de stockage et le nombre de tâches associés.

La figure 7.9 illustre la taille moyenne des données pour un nœud de l'application QAP en fonction de la taille du problème (instance du problème). Comme l'on peut le constater

sur cette figure, la taille des données associées à un nœud de l'application augmente exponentiellement en fonction de la taille du problème.

La figure 7.10 montre (à gauche) la taille totale des données sauvegardées sur le support de stockage durant l'exécution de l'application QAP (NUGENT16S) avec le protocole SEL et le protocole TIC, en utilisant trois périodes différentes $\tau = 1$, $\tau = 5$ et $\tau = 10$ secondes. Cette figure montre aussi (à droite) le nombre total de tâches sauvegardées durant l'exécution. Comme on peut l'observer sur cette figure, la taille des données sauvegardées est directement liée au nombre de tâches sauvegardées. Avec une approche de type TIC le nombre total de tâches sauvegardées, avec une période $\tau = 1s$ est 437 contre 1319 tâches (le nombre total de tâches engendrées par l'application) pour une approche de type SEL. La figure 7.10 montre également que la taille des données sauvegardées avec TIC pour ajouter la tolérance aux pannes peut être contrôlée en augmentant la valeur de la période τ de sauvegarde.

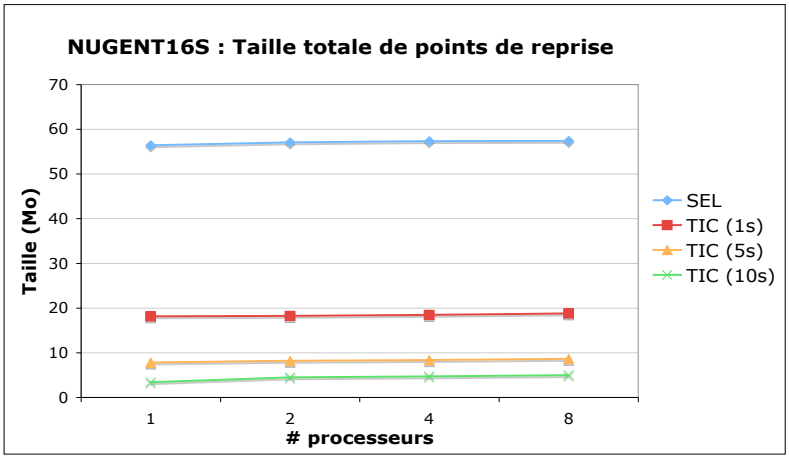

Figure 7.11 – NUGENT16S : Taille totale de points de reprise sauvegardés sur le support de stockage en fonction du nombre de processeurs.

L'influence du nombre de processeurs sur la taille des données sauvegardées par les protocoles TIC et SEL est présentée dans la figure 7.11. Comme on peut le constater sur cette figure, la taille totale des données sauvegardées par les deux protocoles augmente de façon très légère en augmentant le nombre de processeurs. Cette faible augmentation est liée au nombre faible d'opérations de vols réussis.

L'intérêt des résultats présentés dans cette section est la possibilité qu'ils donnent de déterminer, en pratique, la configuration appropriée, par exemple, le nombre de supports de stockage à utiliser pour l'exécution tolérante aux pannes d'une application, Ceci, grâce à la connaissance ou à l'estimation qu'ils permettent de certains paramètres comme, par exemple, le nombre de tâches et la taille moyenne des données par tâche.

7.4.5 Influence du nombre de pannes

L'objectif de cette expérience est de montrer le surcoût introduit par nos protocoles en fonction du nombre de pannes durant l'exécution. Dans cette expérience, nous avons fixé le nombre p de processeurs à 60 et le nombre de supports de stockage à 60 également. Une panne franche est produite aléatoirement par l'envoi d'un signal *kill* à un processus de l'application. Un processus défaillant est remplacé immédiatement par un nouveau processus sur une nouvelle ressource.

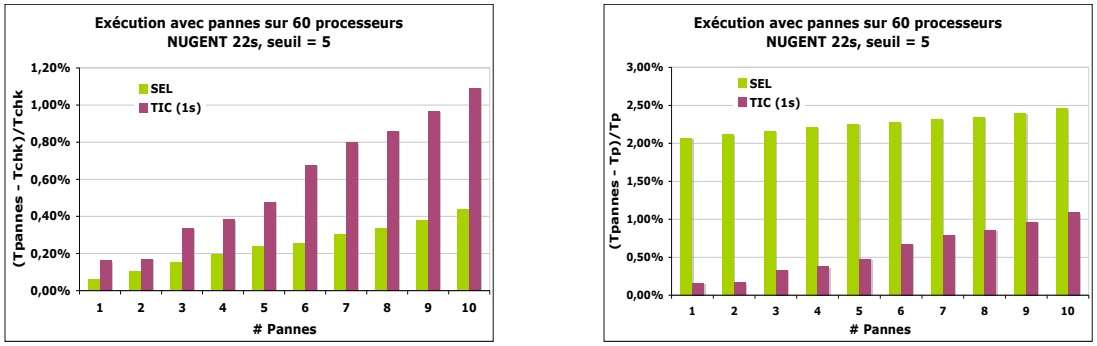

Figure 7.12 – NUGENT22S sur 60 processeurs : Surcoût introduit en fonction du nombre de pannes

L'impact du nombre de pannes sur l'exécution pour les deux protocoles TIC et SEL est montré dans la figure 7.12 : à gauche le surcoût relatif à l'exécution sans panne ($\frac{T_{pannes}-T_{chk}}{T_{chk}}$ où T_{pannes} est le temps d'exécution avec pannes et T_{chk} est le temps d'exécution avec sauvegarde et sans panne) et à droite le surcoût relatif à l'exécution sans protocole de tolérance aux pannes ($\frac{T_{pannes}-T_p}{T_p}$ où T_p est le temps d'exécution sans protocole). En effet, la première remarque est que le surcoût de la reprise augmente linéairement avec le nombre de pannes.

Comme on peut le constater sur la figure 7.12 (à gauche), qui présente uniquement le coût de la reprise en fonction du nombre de pannes, le coût de reprise avec le protocole SEL est moins important qu'avec le protocole TIC : pour une seule panne 0,06% avec SEL et 0,11% avec TIC. Ceci est dû au fait que, la reprise avec SEL consiste à ré-exécuter une seule tâche, le temps moyen d'exécution d'une tâche de calcul est 0,23 secondes, tandis que la reprise avec TIC consiste à ré-exécuter un ensemble de tâches selon la période τ.

La figure 7.12 (à droite) montre le coût total des deux protocoles en fonction du nombre de pannes. On remarque que le coût introduit par TIC est plus petit que celui introduit par SEL.

Ces résultats confirment l'analyse théorique des deux protocoles (§5.5) ; introduire un coût élevé pour la sauvegarde résulte en un coût faible pour la reprise (le cas du SEL) et inversement (le cas du TIC).

7.5 Visualisation de l'arbre du QAP

Dans un objectif d'observer l'évolution de l'arbre (le graphe de flot de données) de l'application QAP à l'exécution. Nous avons utilisé FlowVR [1], développé également au sein du projet MOAIS. C'est un environnement de développement et d'exécution pour les applications interactives haute performance allant de la réalité virtuelle à la visualisation scientifique. À terme, l'objectif d'un tel couplage est de pouvoir interagir avec une application parallèle, en cours d'exécution, en fonction de son comportement à l'exécution.

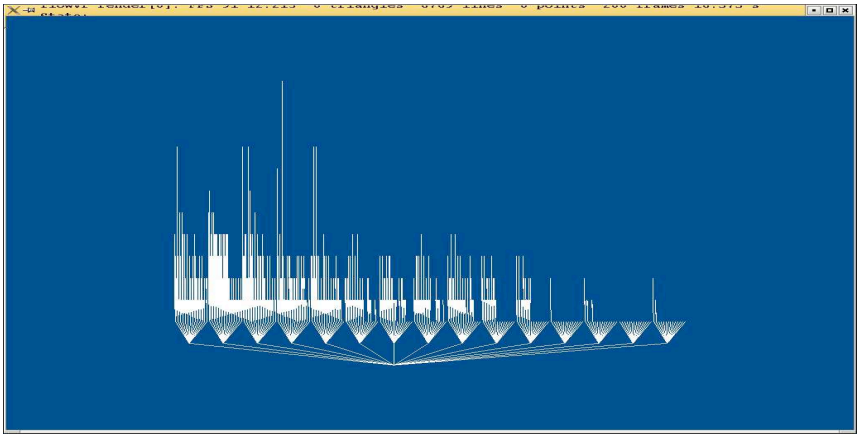

Figure 7.13 – L'arbre du QAP (NUGENT15) : Visualisation avec FlowVR

La figure 7.5 illustre l'arbre de création de tâches pour l'instance NUGENT15 de l'application QAP à la fin de son exécution. Il est à noter que la visualisation de cette arbre avec FlowVR se fait en temps réel.

7.6 Expérimentations sur grille

Dans cette section, nous présentons les expériences menées sur Grid5000. L'objectif de ces expériences est la validation du mécanisme de tolérance aux pannes sur une plateforme hétérogène.

Les expériences que nous présentons ici ont pour objectif la validation de l'hétérogénéité et de la dynamicité sur l'application QAP. Pour cette expérience, nous disposons de 30 processeurs sur trois grappes de Grid5000, 10 processeurs par grappe :

– la grappe “grillon” (Nancy) : 47 bi-AMD Opteron 246 (2 GHz) interconnectés par un réseau Gigabit Ethernet ;
– la grappe “paraci” (Rennes) : 64 bi-Intel Xeon (2.4 GHz) interconnectés par un réseau Gigabit Ethernet ;
– la grappe “tartopom” (Rennes) : 32 bi-Power PC (2 GHz) interconnectés par un réseau Gigabit Ethernet ;

La figure 7.14 montre le temps d’exécution de l’instance NUGENT20s de l’application QAP sur chacune des trois grappes indépendamment, sans et avec le protocole TIC avec une période de sauvegarde $\tau = 5$ secondes et un seul serveur de stockage à l’extérieur de ces trois grappes. Comme l’on peut observer sur cette figure, le temps d’exécution sans sauvegarde varie en fonction de vitesse des processeurs et le temps d’exécution avec le protocole TIC introduit un surcoût près de 5%.

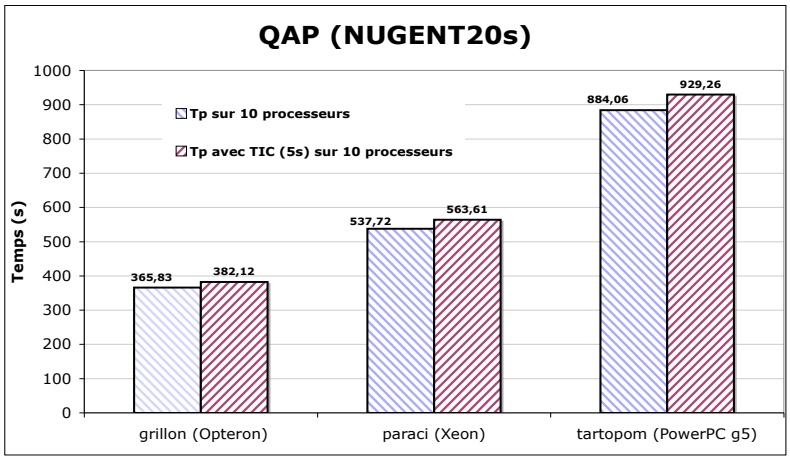

Figure 7.14 – QAP (NUGENT20s) : temps d’exécution sur chaque grappe, 10 processeurs par grappe.

La figure 7.15 illustre les résultats de l’exécution de l’instance NUGENT20s du QAP sur l’ensemble des 30 processeurs. Cette figure montre les résultats de trois expériences sur la grille : la première consiste à mesurer T_p, le temps d’exécution sans mécanisme de tolérance au pannes. La deuxième expérience mesure le temps d’exécution avec le protocole TIC en utilisant une période $\tau = 5s$ et trois serveurs de stockage, un serveur de stockage par grappe, qui se situent sur une quatrième grappe de Grid5000 (“IDPOT” de Grenoble). La troisième expérience consiste à lancer l’exécution, avec la même configuration que la deuxième expérience, on arrête l’exécution après une durée égale à 50% du temps d’exécution de la deuxième expérience et puis on redémarre l’application sur deux grappes seulement (“grillon” et “paraci”).

Les résultats obtenus de ces expériences, illustrés dans la figure 7.15, montre que l’approche choisie pour la tolérance aux pannes permet de restaurer une application sur des machines hétérogènes et sur un nombre variable de processeurs.

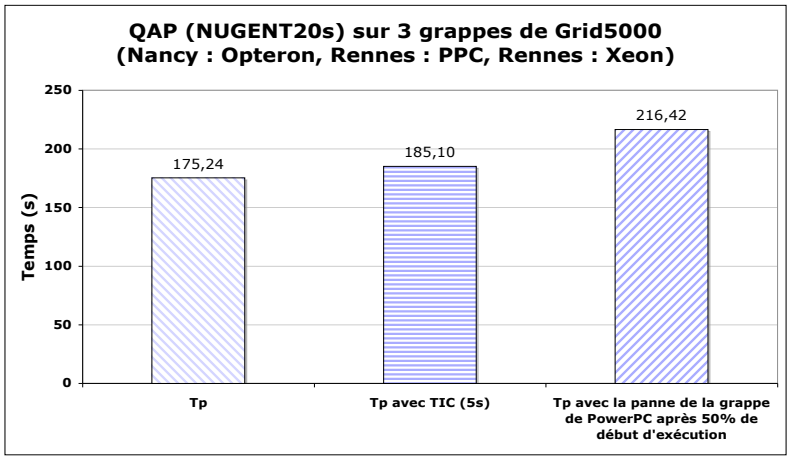

Figure 7.15 – QAP (NUGENT20s) : Exécution avec pannes sur une grille hétérogène.

7.7 Conclusion

Dans ce chapitre, nous avons pu montrer que les deux protocoles de tolérance aux pannes (SEL et TIC) présentés dans le chapitre 5 et leurs implantations dans KAAPI se comportaient correctement pour une application réelle. Les résultats obtenus avec le QAP (*Quadratic Assignment Problem*) sont encourageants et permettent d'envisager une exploitation fiable d'un plus grand nombre de processeurs pour la résolution d'instances de plus grande taille de cette application.

Troisième partie

Applications

Chapitre 8

Adaptabilité et tolérance aux pannes

Sommaire

8.1 Introduction

La durée d'exécution de certains applications parallèles, comme dans le domaine de recherche combinatoire ; par exemple l'application QAP présenté dans le chapitre 7, ne peut être estimée *a priori*. Cette incertitude complique l'exécution de ces applications sur une plateforme donnée :

- sur une plateforme comme Grid5000, avant de lancer l'exécution d'une application, une phase de réservation de nœuds est imposée où il est nécessaire de préciser le nombre de nœuds et la durée d'utilisation de ceux-ci afin de pouvoir lancer l'exécution sur ce type de plateforme. Le nombre de nœuds et la durée d'utilisation sont généralement limités afin de satisfaire les différents utilisateurs.
- sur une plateforme dediée à la production comme celle du CEA, ses responsables allouent un certain nombre de nœuds pendant une durée t et demandent quels seront les résultats que l'on pourra obtenir avec une telle configuration.

Dans ce chapitre, un mécanisme pour l'adaptation d'une exécution parallèle dans les environnements dynamiques et hétérogènes est défini pour les applications de type graphe de flot de données. Notre mécanisme est particulièrement utile pour les calculs multithreadés et massivement parallèles que l'on peut trouver sur les grappes ou sur les grilles de calcul.

En basant l'état d'exécution sur un graphe de flot de données, cette approche montre une grande flexibilité pour l'adaptation d'une application parallèle. Cette adaptation permet de changer le comportement en fonction de plusieurs paramètres pouvant être observés durant l'exécution comme l'efficacité, le nombre de ressources et la qualité de service attendue.

La tendance actuelle dans le calcul parallèle est de pouvoir exécuter des applications parallèles avec dépendances de données sur une plateforme distribuée à grande échelle et hétérogène comprenant des nœuds SMP. Ces plateformes comme les grilles ou encore les systèmes pair à pair sont bien connues pour être hétérogènes et fortement dynamiques en contenu et en charge, c.-à-d. des nœuds peuvent, à tout moment, joindre ou quitter le système.

Concernant les applications parallèles, ce dispositif pose un grand problème pour exploiter entièrement le parallélisme induit par l'application. Dans ce chapitre, nous étudions comment adapter l'exécution des applications parallèles à l'environnement d'exécution afin d'atteindre des rendements élevés en utilisant les mécanismes de tolérance aux pannes proposés dans le chapitre 5.

8.2 Adaptabilité

Ces dernières années, l'adaptation automatique des applications parallèles et réparties a attiré de plus en plus l'attention de la communauté scientifique pour atteindre un rendement élevé lors de l'exécution sur une plateforme dynamique et hétérogène. L'étude des mécanismes pour l'auto-adaptation adapté à de telles plateformes est devenue un secteur actif de recherche [35, 125, 42, 120, 25, 121]. Ces études diffèrent dans le type d'adaptation en ce qui concerne les environnements, le logiciel ou l'exécution du calcul.

Dans [25], les auteurs étudient l'adaptation de la charge de travail des applications à gros grain (i.e composants) sur l'environnement d'exécution. L'approche est basée sur une opération de migration d'un composant. Les points où l'application pourrait migrer sont explicitement fournis par le programmeur de l'application.

Les auteurs de [121] ont étudié l'adaptabilité de la charge de travail des applications à grain fin (i.e. passage de message). Le mécanisme d'adaptation proposé se fonde sur une opération de migration d'un processus. Un contrôleur prend la décision d'adapter l'exécution de l'application à partir d'informations sur le temps d'exécution recueillies à l'aide d'outils de surveillance et d'analyse (fournis par d'autres composants du système).

Notre travail prolonge les deux mécanismes précédents en rendant transparent à l'utilisateur :

1. la définition des points de migration pour une application ;
2. le re-ordonnancement de l'application pour adapter la performance de l'exécution.

L'idée pour résoudre (1) et (2) est basée sur la reflection [25, 86] du système : durant l'exécution, une représentation causalement connectée du système est le graphe de flot de données entre les tâches représentant le futur de l'execution [49]. Cette représentation permet de tenir compte de la structure du calcul, y compris des communications entre les tâches d'une application.

8.3 Adaptabilité à la dynamicité de ressources

KAAPI est un intergiciel qui permet de développer une application de manière à ce que l'ordonnancement soit une boîte noire : changer l'algorithme d'ordonnancement du programme n'exige aucune modification de l'application. L'architecture de l'intergiciel est illustrée sur la figure 8.1. Le point clé sur lequel repose la conception du KAAPI est la représentation abstraite de l'application comme un graphe de flot de données. Cette représentation était initialement choisie parce qu'elle est appropriée aux algorithmes d'ordonnancement. Dans [59, 61] nous proposons l'utilisation de cette représentation pour implanter la tolérance aux pannes.

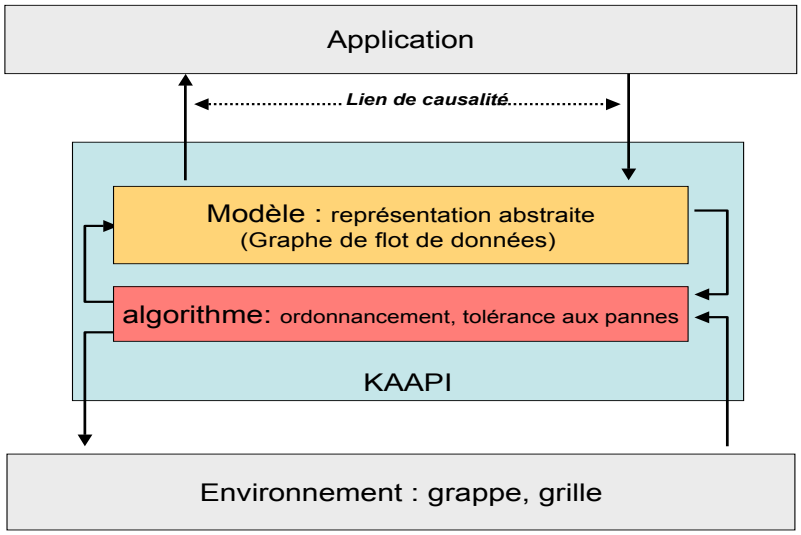

Figure 8.1 – L'architecture de l'intergeciel KAAPI .

Pendant l'exécution d'une application sur une plateforme dynamique, deux changements importants peuvent se produire : le départ et l'arrivée de ressources. Les prochaines sections décrivent le mécanisme d'adaptation pour ces deux cas. L'implantation de ce mécanisme d'adaptation a été réalisée dans la boîte "algorithme" décrite dans l'architecture de KAAPI (figure 8.1).

Dans la section suivante, nous présentons le moniteur de KAAPI qui offre une vision globale de l'exécution parallèle distribuée d'une application. Outre la visualisation de l'exécution, le moniteur de KAAPI est utilisé pour automatiser la procédure d'adaptation afin de rendre le mécanisme d'adaptation transparent à l'utilisateur.

8.3.1 Monitoring dans KAAPI

Afin d'observer l'exécution parallèle distribuée d'une application sur un grand nombre de processeurs, nous avons développé dans KAAPI un mécanisme global de traçage de certains paramètres à l'exécution comme le nombre de requêtes de vol, le nombre de tâches déjà exécutées, la charge *cpu* de la machine, l'efficacité, etc ...

Un contrôleur qui permet l'interaction entre l'utilisateur et l'exécution en cours a été également dévoloppé dans KAAPI. La figure 8.2 présente l'interface graphique qui permet à la fois de visualiser la trace d'exécution et de contrôler cette exécution.

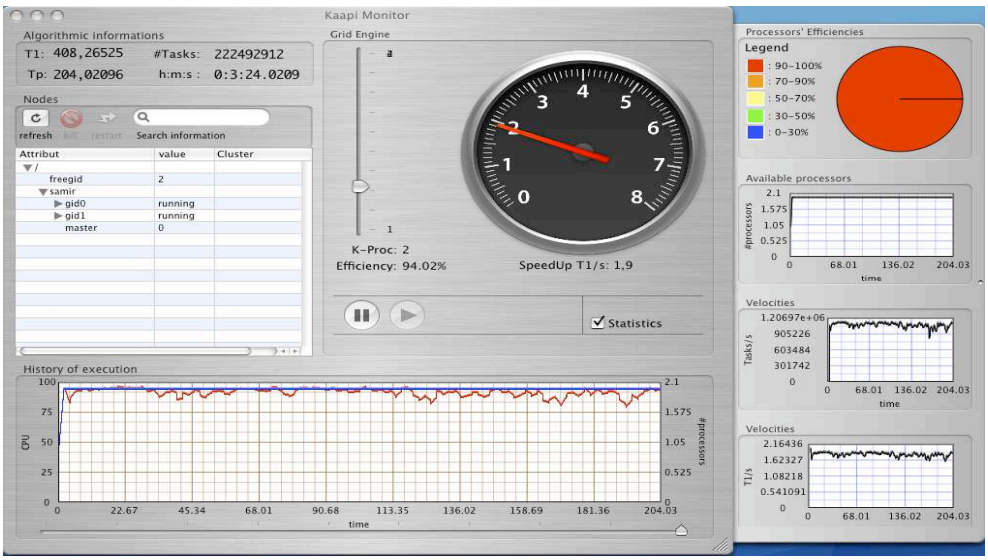

Figure 8.2 – Le moniteur de KAAPI.

8.3.2 Adaptation au départ des ressources

Une classification des départs des ressources peut être la suivante : les départs involontaires et les départs volontaires. Un départ involontaire se produit en cas de défaillance d'une ressource ; dans ce cas l'application n'a aucune information sur la date de départ. Par contre,

dans le cas d'un départ volontaire, l'application possède l'information qu'une ressource va la résilier et peut ainsi le prévoir.

L'idée principale pour traiter les deux cas précédents est de définir un point de reprise pour un processus. Ce point de reprise doit être portable (i.e. indépendant de la plateforme) et ne pas nécessiter d'allouer une nouvelle ressource pour restaurer celle partante à partir de ses points de reprise. Par exemple, un processus sur une ressource, "vivante", peut ajouter les tâches stockées dans un point de reprise à son propre calcul courant. Le mécanisme de tolérance aux pannes est appliqué dans ce cas comme décrit dans [59, 61] qui , en cas de panne d'un nœud, permet de restaurer sur un autre nœud les tâches de l'application qui ne sont pas terminées sur le nœud partant.

En cas de départ volontaire d'une ressource, l'idée de "migration du calcul" est appliquée : juste avant la date du départ, un point de reprise est réalisé. La charge de travail et les données sont ainsi déplacées sur une autre ressource.

8.3.3 Adaptation à l'arrivée des resources

L'adaptation à l'arrivée d'une ou plusieurs nouvelles ressources consiste à ajouter des processus de calcul durant l'exécution de l'application. Le problème posé dans ce cas est le re-ordonnancement de l'application en incluant les nouvelles ressources disponibles. Dans notre cas où l'algorithme d'ordonnancement est un algorithme dynamique de type vol de travail, une fois qu'une nouvelle ressource rejoint l'application, elle commence à voler du travail aux autres ressources déjà existantes, après avoir obtenu le code de l'application.

8.3.4 Adaptabilité dans KAAPI

Concernant l'adaptabilité à la dynamicité de ressources, un mécanisme d'adaptation doit répondre aux questions suivantes :

1. Quel est la méthode de découverte de ressources ?
2. Quel est le mécanisme de détection de départ de ressources ?
3. Comment faire pour que le mécanisme d'adaptation soit transparent pour l'utilisateur (i.e. automatique) ?

Afin de répondre à la première question, nous proposons l'utilisation du système de réservation (batch) disponible sur la plateforme d'exécution cible. L'idée est d'exploiter la fonctionnalité principale offerte par ces systèmes de réservation qui vise à identifier l'état des ressources de la plateforme (i.e. disponible, non disponible, occupée, libre, ...) et à l'attribution de ressources libres à un job soumis.

Concernant la deuxième question, le départ d'une ressource peut être détecté par l'une des deux méthodes suivantes selon le type de départ :

– le départ involontaire de ressources (défaillance de ressources) est détecté par le détecteur de défaillance utilisé par KAAPI ;
– le départ volontaire de ressources correspond à la fin de la durée de la réservation de ces ressources. Ce départ peut être facilement prévu et donc détecté par l'utilisation d'un délai de garde (*timeout*).

La réponse à la troisième question consiste à automatiser :

– la procédure d'ajout de nouvelles ressources à une exécution en cours suite à la découverte de ces ressources ;
– la procédure de reprise de ressources résiliants l'exécution en cours suite à la détection du départ de ces ressources.

8.4 Exemple d'utilisation

Nous présentons, dans cette section, un exemple typique qui illustre l'utilité du mécanisme d'adaptation proposé sur la plateforme Grid5000.

La plateforme Grid5000 est une grille expérimentale d'environ 1496 PCs constituée d'un ensemble de grappes géographiquement réparties sur l'ensemble du territoire français (Bordeaux, Grenoble, Lille, Lyon, Orsay, Rennes, Nancy, Sophia-Antipolis et Toulouse). Le but principal de cette plateforme est de servir d'instrument pour la recherche sur les Grilles et les systèmes P2P.

OAR-GRID [1] [28, 26, 27] est un outil permettant d'effectuer des réservations de machines sur l'ensemble de Grid5000. Le principe de réservation sur grille proposé par OAR-GRID est très simple : il s'agit d'effectuer une réservation sur chacune des grappes souhaitées. Si une tentative de réservation échoue, toutes les réservations sont annulées et l'opération globale également. Un identifiant est associé à chaque réservation ; cet identifiant sert à retrouver toutes les informations liées à l'ensemble des réservations induites sur les grappes. Pour effectuer une réservation OAR-GRID, il faut préciser le nombre de machines, la durée d'utilisation, la date de son début d'exécution ainsi que la nature (prioritaire ou non) de la réservation. Une réservation non prioritaire (également dénomée "besteffort") sera annulée si une réservation prioritaire doit commencer son exécution alors qu'il n'y a pas assez de nœuds libres pour satisfaire la réservation prioritaire. Étant donné qu'OAR-GRID ne fournit pas de techniques de tolérance aux pannes ou de migration, le travail effectué par la réservation annulée est alors perdu.

La durée d'exécution d'une application comme le QAP (voir le chapitre 7) est difficilement estimable [2] pour certaines instances[2]. L'obtention des résultats, pour les grandes instances de cette application, necéssite une exécution sur un grand nombre de nœuds et pour une durée très importante. Comme on ne peut pas estimer cette durée à l'avance, la seule façon d'être certain d'obtenir des résultats est de demander une réservation pour une très longue durée, bien supérieure aux besoins réels. Réserver un aussi grand nombre de nœuds pour une durée aussi longue n'est pas possible en pratique, car cela paralyserait tous les autres utilisateurs de Grid5000.

Le mécanisme d'adaptation de l'exécution parallèle basé sur les techniques de tolérance aux pannes proposé dans cette thèse offre une solution afin de résoudre ce type de problème.

La figure 8.5 illustre l'historique des réservations sur la grappe d'Orsay pour le mois de Janvier 2006. Une case de couleur blanche signifie que le nœud associé est libre et une case d'une autre couleur signifie que le nœud est occupé. Chaque couleur correspond à une réservation. Comme on peut le constater sur la figure 8.5, il y a un nombre important de nœuds qui n'étaient pas utilisés durant certaines périodes de temps. Nous nous sommes intéressés à l'utilisation de ces périodes d'inactivité pour exécuter une application de longue durée

[1] https ://helpdesk.grid5000.fr/wiki/index.php/OAR_GRID
[2] http ://www.seas.upenn.edu/qaplib/inst.html

sans gêner les autres utilisateurs. Pour ce faire, nous proposons d'utiliser notre mécanisme d'adaptation, en le couplant au mécanisme de réservation non prioritaire d'OAR-GRID. Cela permet d'ajouter les nœuds qui deviennent disponibles à une exécution déjà lancée, tout en garantissant qu'ils seront relâchés quand une autre réservation prioritaire devra commencer.

8.5 Experimentations

Les objectifs de ces expériences sont de valider l'implantation, dans KAAPI , de l'adaptation à l'arrivée et au départ de ressources en utilisant l'application de test UB_Tree_{F*} présentée dans le chapitre 6.

Les deux expériences, présentées dans cette section, ont été effectuées sur la grappe de Nancy de Grid5000, les paramètres de l'application UB_Tree_{F*} utilisés sont $n = 58$, $k = 2$ et le seuil d'arrêt du parallélisme $s = 25$. Cette grappe est composé de 32 biprocesseurs Opteron (2 GHz) interconnectés par un réseau Fast Ethernet 100 Mo/s.

Le temps d'exécution séquentielle de l'application est $T_s = 6605$ secondes, le temps d'exécution parallèle sur un seul processeur est $T_1 = 6617$ secondes et le temps d'exécution sur $p = 40$ processeurs est $T_p = 179$ secondes.

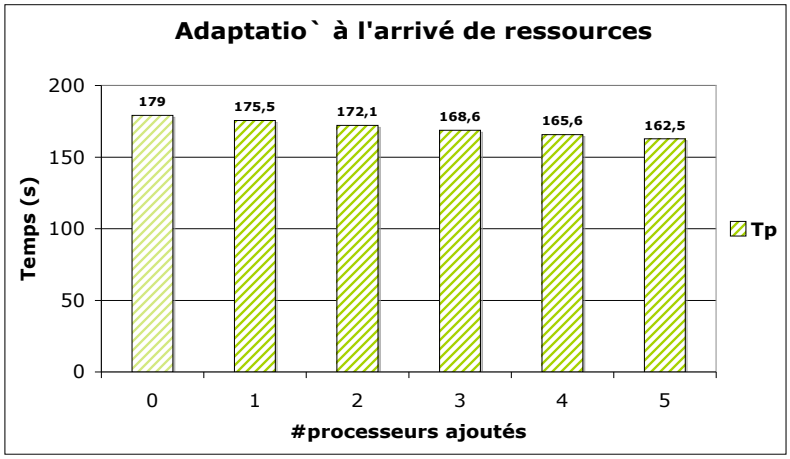

Figure 8.3 – $UB\_Tree_{F*}(58, 2, 25)$: Temps d'exécution en fonction du nombre de nouveaux processeurs rajoutés au calcul.

La Figure 8.3 montre les temps d'exécution en fonction du nombre de nouvelles ressources rejoignant le calcul ; au départ l'exécution est lancée sur 40 processeurs de la grappe, chaque barre sur la figure correspond à une exécution avec i ($i = 0, ..., 5$) nouveaux processeurs rajoutés au calcul. Un nouveau processeur est ajouté toutes les 20 secondes à compter du début de l'exécution. Comme on peut l'observer sur cette figure, le temps d'exécution diminue linéairement en fonction du nombre de processeurs rajoutés. L'efficacité observée de l'utilisation des ressources est $> 90\%$.

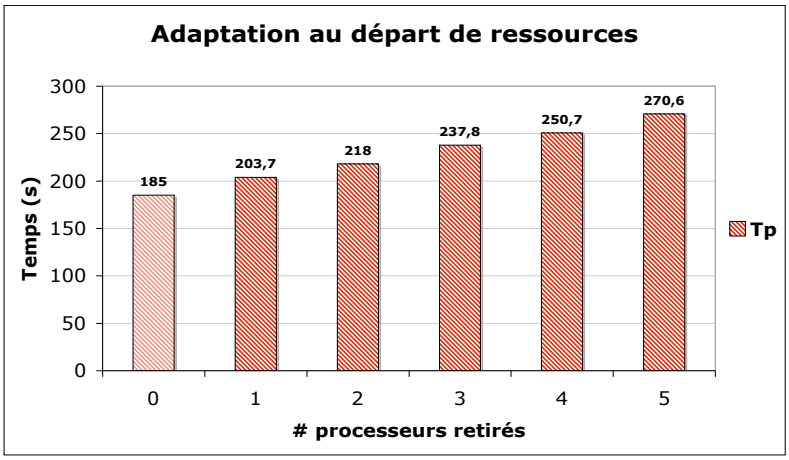

Figure 8.4 – $UB\_Tree_{F*}(58, 2, 25)$: Temps d'exécution en fonction du nombre de processeurs retirés du calcul.

La deuxième expérience illustrée sur la figure 8.4 présente l'impact du mécanisme d'adaptation sur les départs de ressources. Le temps d'exécution de l'application de test UB_Tree_{F*} avec les paramètres $n = 58$, $k = 2$ et $s = 25$ et le protocole TIC en utilisant une période de sauvegarde $\tau = 10$ secondes et un seul support de stockage (serveur centralisé de checkpoints) sur un processeur est $T_1 = 6909$ secondes et sur $p = 40$ processeurs est $T_p = 185$ secondes. Chaque barre sur la figure 8.4 correspond à une exécution de l'application avec i ($i = 0, ..., 5$) processeurs retirés du calcul : un processeur est retiré toutes les 20 secondes à compter du début de l'exécution par l'envoi d'un signal *kill* à un processus. Une fois le processus retiré, on signale à un autre processus de l'application de restaurer le point de reprise du processus retiré. Comme on peut l'observer sur cette figure, le temps d'exécution augmente linéairement en fonction du nombre de processeurs retirés.

8.6 Conclusion

Afin d'adresser la performance d'applications parallèles, nous avons présenté un mécanisme d'adaptation pour les applications parallèles de type flot de données sur une plateforme hétérogène et dynamique.

L'état de l'application parallèle est représenté de façon portable (représentation abstraite) en utilisant un graphe de flot de données. Cet état est le premier objet sur lequel l'algorithme d'adaptation agit. En raison de la représentation causalement connectée entre le graphe de flot de données et l'application, une modification de cette représentation entraîne une modification du comportement de l'exécution.

Cette représentation abstraite est une description, indépendante de la plateforme, de l'état de l'application qui permet de s'adapter à une grappe ou une grille dynamique et hétérogène. Le surcoût associé à ce mécanisme est faible, au point d'être négligeable en cas d'ajout

de ressources pour des applications ayant un chemin critique très faible et si le nombre de processus n'est pas très important.

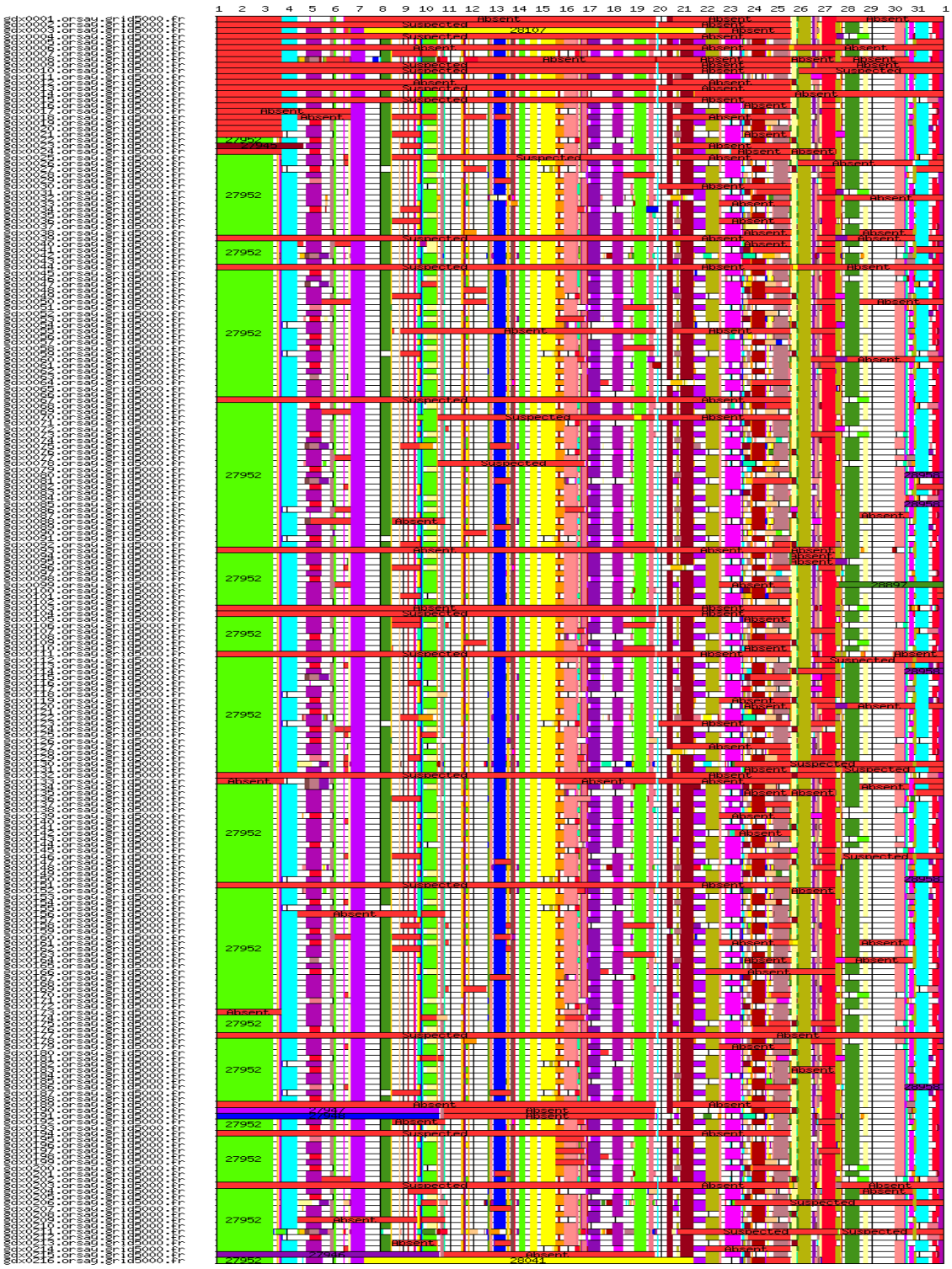

Figure 8.5 – L'historique du mois de Janvier 2006 de la réservation sur la grappe d'Orsay de Grid5000.

Chapitre 9

Certification du calcul des applications parallèles

Sommaire

9.1 Introduction

Ce chapitre a pour objectif de montrer une application directe du protocole proposé SEL à la résolution du problème de la certification de programmes massivement parallèles (c'est à dire possédant un grand nombre n de tâches) sur une plateforme distribuée de grande taille, type grille ou pair-à-pair.

Les pannes traitées dans le cadre de cette thèse sont, pour l'instant, de type pannes franches ; cette classe de pannes suppose que le processus affecté par une panne arrête totalement de délivrer son service et que la panne est détectée par les autres processus participant à l'exécution. Mais que se passe-t-il si un processus commence à délivrer des résultats erronés ? Dans ce cas, la panne survenue durant l'exécution est invisible et les résultats finaux de l'exécution sont incorrects ce qui est très grave pour certaines applications d'aide à la décision notamment dans le domaine médical.

Le cas où un processus commence à produire des résultats erronés est connu sous le terme [79] "processus affecté par une panne par valeur". La nature d'une telle panne peut être intentionnelle et la panne est créée avec une intention malicieuse ce qui est appelé "attaque malicieuse".

Un exemple typique d'une telle attaque est un virus, propagé par un ver et affectant un type de système d'exploitation donné en permettant à un attaquant de contrôler les ressources affectées. Les statistiques de CERT/CC [31] témoignent de l'impact des attaques massives : tandis que le nombre des vulnérabilités rapportées se stabilise : 4129 en 2002, 3784 en 2003, le nombre d'incidents rapportés croît exponentiellement : 82094 incidents rapportés en 2002 contre 137529 en 2003.

Deux problèmes sont à résoudre afin de tolérer ce type de pannes : il faut d'abord détecter la panne et puis la corriger. Nous nous intéressons dans ce chapitre à la détection de pannes par valeur en utilisant le protocole SEL.

Dans le cadre de ce travail, nous considérons une application parallèle modélisée par un graphe de tâches. Nous supposons que l'application tolère un grand nombre d'erreurs, typiquement $\delta.n$ fautes où n est le nombre de tâches de l'application et δ son taux de tolérance. Cette hypothèse est vérifiée par de nombreuses applications cible des plateformes de calcul global [55, 105] qui sont souvent de type simulation Monte-Carlo. Pour assurer un tel taux de tolérance lorsque les tâches sont indépendantes, L. Sarmenta propose dans [105] une technique basée sur le calcul de la crédibilité pouvant être accordée à un résultat donné.

Cette approche reste limitée car elle suppose, d'une part, que l'utilisateur détermine la crédibilité qu'il accorde à chaque composant impliqué dans le calcul et, d'autre part, que les tâches affectées par les attaques suivent une distribution de Bernouilli. Dans le cadre de tâches avec dépendances, L. Gao et G. Malewicz [51] utilisent des techniques de duplication et d'exécution de certaines des tâches sur des ressources fiables (toujours supposées peu nombreuses) qui permettent d'atteindre un taux de tolérance δ.

Ce taux de tolérance est en particulier vérifié par l'application médicale étudiée dans le projet Ragtime [91, 64] qui motive ce travail. Dans cette application, une image médicale est soumise pour comparaison (analyse de similarité) aux images stockées dans différentes bases comportant un grand volume de données radiographiques. Dans un premier temps, les images liées à la requête soumise sont extraites ; puis les calculs de similarité sont effectués sur chaque image extraite ; enfin, les résultats sont classés en fonction de leur similarité. Il y a deux niveaux de parallélisme : entre les calculs de similarité d'une part et au sein d'un même

calcul qui peut lui-même porter sur une séquence d'images. Cette application correspond à un arbre de tâches. Plus généralement, nous considérons que l'application est modélisée par un graphe de dépendances (flot de données macroscopique), implicite à de nombreux langages de programmation parallèle comme KAAPI .

Étant donnés les résultats d'une exécution, le problème est alors de déterminer si ils sont corrects (le taux d'erreur est alors garanti inférieur à δ) ou si l'exécution a été victime d'une attaque massive qui a conduit à la falsification des résultats de n_F tâches.

Pour résoudre ce problème lorsque les tâches sont indépendantes, C. Germain et N. Playez [55] considèrent le cas où la plupart des ressources utilisées sont "honnêtes" alors que les ressources corrompues falsifient toujours leurs résultats. En supposant une distribution de Bernouilli des erreurs, les auteurs proposent l'utilisation d'un test statistique séquentiel pour se ramener à la vérification d'un nombre limité de tâches indépendantes choisies aléatoirement et re-exécutées sur des ressources fiables. Ce test prend en compte deux paramètres fixés par l'utilisateur : le risque de fausse alarme (faux positif) et de non détection (faux négatif).

De façon similaire à [55], nous considérons une borne inférieure (et non supérieure comme pour [51, 105]) sur le nombre de tâches falsifiées ce qui permet d'inclure les attaques classiques qui peuvent affecter tout un sous-réseau (ver, attaques basées sur une plage d'adresses IP etc...).

9.2 Définitions et hypothèses

9.2.1 Plateforme d'exécution

L'exécution de l'application considérée repose sur une plateforme distribuée (de type grille ou pair-à-pair). Cette plateforme se divise en deux types de ressources : les *workers* d'une part, considérés comme non sûrs car exécutant les tâches dans un environnement peu fiable et les *oracles* d'autres part qui sont des ressources sûres utilisées pour certifier les résultats de l'exécution. Les communications entre ces ressources sont effectuées par un serveur de points de reprise ("checkpoint") considéré comme sûr et sécurisé qui stocke le graphe de l'exécution (voir paragraphe suivant).

9.2.2 Représentation des exécutions

Dans la suite, l'application parallèle est modélisée par un DAG biparti $G = (\mathcal{V}, \mathcal{E})$. $\mathcal{V}$ est un ensemble de nœuds divisés en deux classes : les tâches, d'une part, et les paramètres (d'entrée et de sortie) de ces tâches, d'autre part. Une tâche t_j est vue au sens de l'ordonnancement de tâches, autrement dit t_j dénote la plus petite unité de programme dans une exécution donnée. $\mathcal{E}$ est un ensemble d'arêtes e_{jk}, $j \neq k$, représentant les précédences entre les nœuds de G.

Le graphe est supposé déterministe et connu de l'utilisateur *a priori*. Enfin, le nombre total de tâches t_j dans G est n.

9.2.3 Impact des pannes par valeur

On note E l'exécution d'un programme G sur un ensemble de ressources non fiables à partir de l'ensemble des paramètres d'entrée initiaux $\hat{I}$. Chaque tâche t de E s'exécute sur les paramètres d'entrée $i(t, E)$ et engendre les résultats $o(t, E)$. $i(t, E)$ est constitué soit de données de $\hat{I}$, soit de sorties d'autres tâches t_k (i.e $o(t_k, E)$). Soit $\hat{E}$ l'exécution du même programme G mais sur un ensemble de ressources fiables.

Définition 11 (Etat d'une exécution) *Une exécution E est dite "correcte" et on note $E = \hat{E}$ si toute tâche utilise les mêmes paramètres d'entrées et engendre les mêmes sorties dans E et $\hat{E}$. Dans le cas contraire, on dit que l'exécution est "falsifiée".*

On aura noté que l'on utilise la notation "chapeau" (par exemple $\hat{I}$ ou $\hat{E}$) pour qualifier un ensemble sûr de valeurs de paramètres ou une exécution effectuée sur des ressources fiables. Ainsi, soit t une tâche et $i(t, E)$ ses paramètres d'entrées lors de l'exécution E sur des ressources non fiables, les paramètres d'entrées de la même tâche t exécutée avec les mêmes entrées mais sur une ressource fiable sont notés $\hat{i}(t, E)$. Enfin, les paramètres d'entrés de la même tâche t dans l'exécution correcte $\hat{E}$ sur des ressources fiables sont notés $\hat{i}(t, \hat{E})$.

De façon similaire, on définit les notations $o(t, E)$, $\hat{o}(t, E)$ et $\hat{o}(t, \hat{E})$ relatives aux sorties de l'exécution de la tâche t.

Une tâche peut être victime d'une attaque, par exemple sous forme d'un virus, d'une exécution d'un client incorrect, ou d'une intrusion lors de la lecture ou l'écriture sur le serveur de points de reprise. Il est important de noter qu'on ne peut pas décider si une tâche a été victime d'une attaque si l'on ne connait que ses paramètres d'entrée et de sortie. Une tâche qui est correcte mais qui a été victime d'une attaque ne pose pas de problème et n'est pas considérée dans la suite comme falsifiée.

9.3 Lien entre le protocole SEL et la certification

Puisque le protocole SEL se base sur la sauvegarde de chaque tâche du programme (du graphe de flot de données) dans un format portable (indépendant de la plateforme d'exécution), il fournit la possibilité de ré-exécuter une seule tâche à partir d'un fichier de journalisation sans contrainte lié aux conditions particulières de la première exécution comme par exemple le type et l'architecture des processeurs.

La méthode de base utilisée généralement pour détecter les pannes par valeur consiste à introduire une technique de redondance. Mais pour pouvoir détecter les pannes par valeur, c'est à dire certifier l'exécution d'une application parallèle, plusieurs questions sont à élucider :

- La première question qui se pose, concerne la méthode de redondance qui peut-être utilisée dans le contexte de calcul parallèle massif avec dépendances de données.
  Pour répondre à cette question et comme on l'a signalé dans le chapitre 2 (§2.6), dans les applications parallèles où la durée du calcul est très importante, toutes les ressources disponibles doivent être utilisées pour le calcul lui-même. Par conséquent, la redondance par duplication spatiale ou temporelle n'est pas adaptée dans ce contexte, d'où l'utilisation d'une redondance informationnelle par un mécanisme de sauvegarde ou de journalisation qui est plus efficace.

– La deuxième question concerne le critère qui permet de déterminer si une exécution est ou non falsifiée. La réponse à cette question dépend d'une part de l'application parallèle elle-même et d'autre par le niveau de sécurité demandé par l'utilisateur. Pour les applications qui ont un taux de tolérance nul, la certification est impossible sauf à exécuter ces applications sur des ressources fiables. Dans [71, 70], nous avons défini les applications cibles de ce travail, l'exécution correcte d'une application (voir §9.2.3) et nous avons proposé un algorithme probabiliste qui détermine si l'exécution est correcte ou non ; cet algorithme est basé sur la re-exécution sur ressources fiables d'un certain nombre de tâches tirées au hasard parmi l'ensemble des tâches de l'application considérée. Le nombre de tâches à vérifier dépend de la probabilité d'erreur fixée.
– La troisième question qui se pose, porte sur la condition permettant de déterminer si une tâche d'un programme avec dépendance a été ou non falsifiée.
Dans le cas d'un programme parallèle composé de tâches indépendantes, on peut déterminer si une tâche a été ou non falsifiée par la simple re-exécution de cette tâche sur une machine fiable. En effet, toutes les tâches étant indépendantes, le résultat d'une tâche n'a aucune influence sur celui d'une autre.
Mais, dans notre cas où les applications considérées ont des dépendances de données la simple re-exécution d'une tâche sur une machine fiable ne suffit pas. Pour comprendre la différence entre ces deux cas, nous allons donner un exemple pour chacun d'eux :

1. **Tâches indépendantes** : La figure 9.1 montre l'exécution sur une machine fiable d'un programme parallèle très basique composé de 5 tâches indépendantes. L'exemple d'une exécution, du programme de la figure 9.1 contenant des pannes par valeur est illustré dans la figure 9.2.

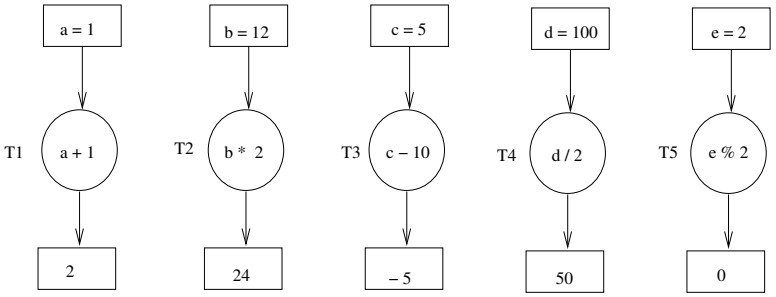

Figure 9.1 – Exécution correcte d'un programme composé de tâches indépendantes.

Comme le montre la figure 9.2, la panne par valeur affectant la tâche T_1 (resp. T_4) n'a aucun impact sur les résultats des autres tâches T_2, T_3 et T_5. En effet, la re-exécution du code correct (le code dans la figure 9.1) de la tâche T_1 (resp. T_4) à partir de ses paramètres sur une machine fiable permet de savoir si T_1 (resp. T_4) a été falsifiée ou non en comparant le résultat de l'exécution de T_1 (resp. T_4) sur machine non fiable (figure 9.2) avec son résultat sur machine fiable (figure 9.1).

2. **Tâches avec dépendances de données** : L'exécution sur une machine fiable d'un programme parallèle simple composé de 5 tâches avec dépendances de données est montrée dans la figure 9.3 : la tâche T_4 (resp. T_5) dépend des résultats des tâches T_1 et T_2 (resp. T_2 et T_3).
La figure 9.4 illustre l'exécution du même programme dans la figure 9.3, mais

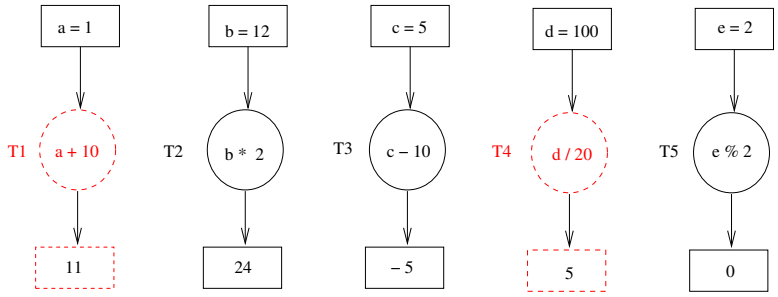

Figure 9.2 – Exécution non correcte d'un programme composé de tâches indépendantes.

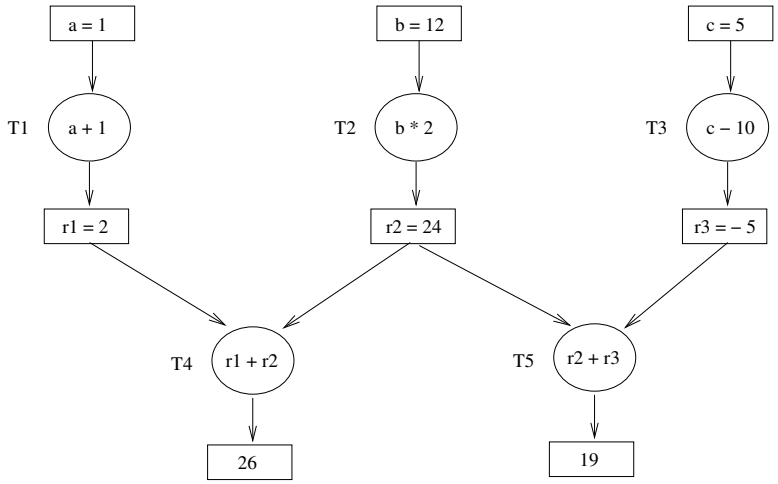

Figure 9.3 – Exécution correcte d'un programme composé de tâches avec dépendances de données.

sur une machine non fiable. La tâche T_3 a été victime d'une attaque et donc a délivré un résultat erroné.

Pour vérifier cette exécution, il faut vérifier un certain nombre de tâches du programme en les re-exécutant sur une machine fiable (le nombre de tâches à ré-exécuter dépend de la méthode choisie pour la certification [70]). Le problème ici est la façon de déterminer le test de vérification d'une tâche. Imaginons que la tâche T_5 a été sélectionnée pour être vérifiée, la re-exécution de T_5 à partir de ses paramètres $r_2 = 24$ et $r_3 = 4$ sur machine fiable donne aussi le résultat 28 qui est égal à son résultat dans l'exécution incorrecte (voir figure 9.4) et donc le résultat de ce test est que la tâche T_5 a été exécutée correctement bien que son résultat soit faux. Pour éviter ce problème, il faudrait assurer que la re-exécution d'une tâche sur la machine fiable soit effectuée à partir de paramètres corrects (les paramètres que la tâche reçoit dans une exécution correcte) [71]. Pour avoir les paramètres corrects d'une tâche à vérifier, il faut les reproduire sur la machine fiable en re-exécutant toutes les tâches dans le graphe des prédécesseurs de la tâche en question [71, 70].

Le protocole de journalisation proposé (SEL) fournit un outil qui répond aux différentes questions posées précédemment [64] : le SEL permet de garder l'histoire de l'exécution parallèle d'un programme si on ne libère pas le journal des tâches terminées. Sa fine granularité

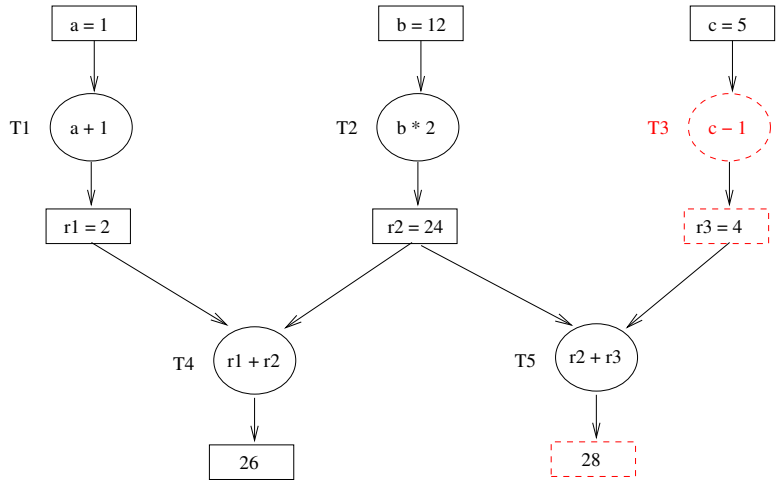

Figure 9.4 – Exécution non correcte d'un programme composé de tâches avec dépendances de données.

permet la re-exécution d'une seule tâche. Le SEL supporte l'hétérogénéité et donc à partir d'un fichier de journal, une tâche peut être re-exécutée sur n'importe quelle autre machine ce qui permet sa vérification sur une machine fiable différente de la machine où s'est exécuté le programme. Le SEL peut fournir la possibilité de reconstruire le graphe des prédécesseurs[1] d'une tâche grâce aux identifications uniques des nœuds du graphe de flot de données (§5.4.2.2).

9.4 Architecture de certification basée sur le protocole SEL

La figure 9.5 illustre la plateforme de calcul global pour la certification d'un programme parallèle distribué : l'utilisateur lance l'exécution de son application, avec le protocole SEL sans libération du journal associé aux tâches terminées, sur la plateforme d'exécution composée d'un ensemble de machine de calcul. Les différents "workers" stockent leurs fichiers de journalisation, réalisés par le protocole SEL, sur un support stable de stockage (Checkpoint Server). Ce support de stockage peut être distribué sur plusieurs machines fiables. Un ensemple de machines sécurisées (oracles) sont reliées de manière sécurisée avec le support stable de stockages. Ces oracles ont pour rôle, la ré-exécution (test) de tâches désignées par l'algorithme de certification, l'allocation et la libération de la place mémoire requise par le protocole SEL.

9.5 Conclusion

Ce chapitre illustre l'application du protocole de journalisation d'un graphe de flot de données (SEL) à la résolution du problème de la certification d'application largement distribuées et s'exécutant dans un environnement hostile (détection de pannes par valeur). Dans

[1] le prédécesseur d'une tâche t est sa tâche mère et l'identifiant de la mère est directement accessible à partir de l'identifiant de la tâche fille t.

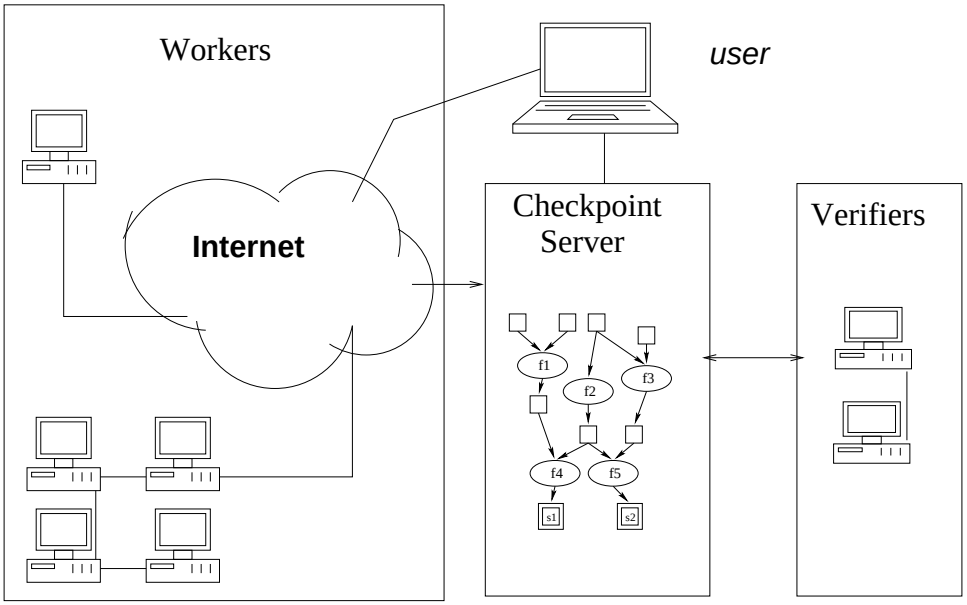

Figure 9.5 – Plateforme de calcul global pour la certification.

ce contexte, les tâches qui composent l'application ou les données qui contiennent les paramètres (d'entrée ou de sortie) de ces tâches peuvent avoir été manipulées par un attaquant.

Chapitre 10

Conclusions

Sommaire

10.1 Rappel des objectifs

Les travaux menés durant cette thèse concernent l'étude, la proposition et la réalisation de mécanismes de tolérance aux pannes pour les applications parallèles s'exécutant dans des environnements hétérogènes de grand taille.

La base de notre travail est un modèle de programmation et d'exécution pour les applications parallèles de type graphe de flot de données. Ce choix a été fait pour les raisons suivantes :

- la représentation de l'exécution par un graphe de flot de données a été utilisée dans de nombreux systèmes car elle modélise bien une exécution distribuée en permettant de représenter les communications qui devront être réalisées à l'exécution ;
- cette représentation permet d'implanter un mécanisme d'exécution "multithreadé" qui active au plus tôt les tâches pour lesquelles toutes les données en entrée ont été produites ;
- cette répresentation est indépendante du nombre de ressources ;
- le graphe de flot de données permet de définir l'état d'exécution comme une structure de données pouvant être communiquée entre ressources hétérogènes, indépendamment du flot de contrôle de l'exécution d'un processus.

Notre objectif était de proposer les services nécessaires pour qu'un tel modèle permette l'exploitation fiable et efficace des plateformes de grande taille connues pour leur hétérogénéité et leur dynamisme.

10.2 Bilan et évaluation de la réalisation

La représentation de l'état de l'exécution d'un programme parallèle par un graphe de flot de données est exploitée, dans ce travail, pour résoudre les problèmes liés à la dynamicité des plateformes considérées. La définition de formats portables pour la représentation des nœuds du graphe a été réalisée afin de résoudre les problèmes d'hétérogénéité. La sauvegarde du graphe de flot de données d'une application durant son exécution sur une plateforme, constitue un point de reprise pour cette application. Par la suite, une reprise de l'exécution de l'application est possible sur un autre type de plateforme à partir de ce point de reprise.

Le mécanisme de tolérance aux pannes que nous proposons repose sur la sauvegarde du graphe de flot de données d'une application parallèle sur un support stable de stockage. Nous proposons deux protocoles de sauvegarde l'un systématique et l'autre utilisant le vol de travail.

Une analyse théorique et un modèle de coût algorithmique de chacun des deux protocoles proposés ont été effectués. La validation expérimentale des deux protocoles est faite sur des applications développées en ATHAPASCAN et KAAPI . En particulier, nous l'avons appliqué à une application d'optimisation combinatoire dans le cadre de l'ACI Grille DOCG et de IMAG-INRIA AHA en collaboration avec le laboratoire PRISM à Versailles, Gilco à Grenoble et LIFL à Lille.

Nous avons également proposé une étude sur l'utilisation de la sauvegarde et de la reprise du graphe de flot de données comme une solution possible pour l'adaptation de l'exécution d'une application parallèle, en fonction des ressources disponibles sur la plateforme et/ou du comportement de l'algorithme exécuté. L'objectif est de garantir l'efficacité et les performances de l'application pendant son exécution en fonction des ressources disponibles à

chaque instant.

Enfin, nous avons proposé, l'utilisation de la technique de sauvegarde / reprise du graphe de flot de données pour certifier l'exécution d'une application parallèle, répartie sur une architecture de grande taille, pouvant potentiellement être victime d'attaques massives.

10.3 Perspectives

Cette contribution propose les extensions et les services nécessaires pour qu'un modèle de programmation de type graphe de flot de données permette une exécution fiable et efficace d'application parallèle de ce type. Elle constitue une base de travail qui peut être complétée dans diverses directions.

Une première direction concerne l'implantation d'un troisième protocole pour la sauvegarde de points de reprise de type sauvegarde coordonnées ; cette méthode de sauvegarde est très efficace pour les applications itératives de type simulation numérique [129, 22]. Le principe de ce protocole appelé CCK (Coordinated Checkpointing in KAAPI) est présenté dans [12]. Le protocole CCK est une amélioration du mécanisme de sauvegarde coordonnée standard car, en cas de panne, uniquement la reprise locale des processus défaillants est requise.

Une seconde direction est l'étude et l'implantation d'un support stable de stockage sur des machines volatiles. En effet, dans le cadre de cette thèse, la machine qui possède un support de stockages (serveur de points de reprise) est considérée fiable.

Une troisième direction, enfin, concerne la définition et la mis en oeuvre d'un méta-protocole (protocole adaptatif) pour la tolérance aux pannes qui permette de basculer d'un protocole à un autre en cours d'exécution en fonction du comportement de l'application à l'exécution, d'une part, et du comportement de plateforme informatique, d'autre part.

Annexes

Annexe A

Analyse spécialisée de coût pour la tolérance aux pannes

Sommaire

A.1 Introduction

Dans ce chapitre, nous étudions une extension du modèle de coût d'exécution des applications KAAPI en tenant compte du coût nécessaire pour offrir la fonctionnalité de tolérance aux pannes. L'ordonnancement considéré ici est un ordonnancement à base de vol de travail.

L'architecture considérée est supposée homogène, le temps d'exécution d'une application sur p processeurs est donc $T_p \leq c_1 T_1/p + c_\infty T_\infty$ (§4.4). Nous noterons par σ le nombre de tâches créées au cours de l'exécution. Dans un premier temps, nous supposerons que le support stable est géré par une machine centrale dont les accès en lecture ou écriture sont séquentiels. Le temps d'un accès élémentaire au support stable est noté t_s.

A.2 Modèle de (sur)coût sans panne

L'objectif de cette section et de présenter le surcoût à l'exécution des deux approches retenues pour la gestion du mécanisme de sauvegarde (TIC et SEL). La section A.3 présente le modèle de coût d'une exécution avec une panne.

A.2.1 Modèle de coût avec SEL

Chaque modification dans le graphe de flot de données induit une écriture sur le support stable : chaque création et suppression de tâche, chaque opération de vol et écriture des données partagées. Ces opérations sont exécutées lors de l'exécution normale du programme ; d'après le principe 1 du *travail d'abord* (§4.4), nous pouvons écrire que le temps d'exécution du programme avec la fonctionnalité de sauvegarde est, sous l'hypothèse de séquentialité d'accès au support stable :

$$T_p{}^* \leq c_1{}^* T_1/p + c_\infty^* T_\infty + O(\sigma t_s)$$

Pour des programmes très parallèles où T_∞ peut être négligé devant T_1/p, d'où le rapport $\gamma^* = T_p^*/T_p$:

$$\gamma^* \leq 1 + p\frac{t_s}{c_1} O(\frac{\sigma}{T_1}) \tag{A.1}$$

Or dans l'implantation de KAAPI , de même que celle de Cilk, $c_1 \simeq 1$. La réduction du surcoût de la sauvegarde dans le cas des programmes à grain fin peut être abordée à deux niveaux : au niveau algorithmique, il convient de réduire le nombre de tâches en diminuant σ ; au niveau de l'implantation, il est important d'utiliser un réseau rapide afin de diminuer t_s et si possible d'utiliser un support stable distribué afin de supprimer la linéarité en p due à l'exécution en séquence des accès. Le surcoût effectif de cette méthode de sauvegarde par journalisation des événements de construction du graphe dépendra des valeurs de t_s et du nombre de tâches crées.

A.2.2 Modèle de coût avec TIC

Dans le cadre de la sauvegarde induite par le vol de travail, chaque processus enregistre son graphe de flot de données sur le support stable toutes les τ secondes. La taille de ce

graphe est liée à la profondeur des appels récursifs, soit $O(T_\infty)$. Le coût de l'opération de sauvegarde d'un processus est $\tilde{t_s} = O(T_\infty t_s)$.

Le nombre de processuss actifs par processeur est au plus 1. Le nombre de sauvegardes réalisées par processus durant son exécution est inférieur ou égal à $\frac{T_p}{\tau}$ et le nombre de sauvegardes forcées dues aux opérations de vols est de l'ordre de $O(T_\infty)$ par processus. En conséquence, le nombre total de sauvegardes N réalisées par chaque processus est :

$$N = O\left(\frac{T_p}{\tau} + T_\infty\right)$$

Le temps d'exécution $T_p^\sharp$ sur p processeurs tenant compte de la sauvegarde induite par le vol de travail est donc : ${T_p}^\sharp = T_p + N\tilde{t_s}$. Le rapport $\gamma^\sharp = T_p^\sharp / T_p$ est égal à :

$$\gamma^\sharp \leq 1 + O(\frac{1}{\tau} + \frac{T_\infty}{T_p})\tilde{t_s}$$

Dans le cas des applications très parallèles $T_\infty \ll T_1$ et pour un nombre borné de processeurs $T_\infty \ll T_p$.

La réduction du coût de la sauvegarde est alors directement liée à la période de sauvegarde. Il est à noter que la période de sauvegarde peut facilement être amortie [94].

A.3 Prise en compte du coût de la reprise

Nous considérons d'abord le cas où la restauration est globale, le temps d'une exécution comprenant une panne se décompose en trois parties : le temps d'exécution considérée comme sans panne, le temps de détection de la panne et le temps d'exécution de la reprise comprenant le temps nécessaire à ré-exécuter le programme jusqu'au point précédant la panne. Notons par T_p^f le temps d'exécution sur p processeurs avec K_f pannes. Nous avons donc $T_p^f = T_p + K_f(T_p^d + T_p^r)$ où T_p^d est le temps de détection et T_p^r le temps de la restauration du processeur fautif.

Nous ferons l'hypothèse d'un détecteur parfait qui est capable de redémarrer un processeur dès que celui-ci tombe en panne, soit $T_p^d = 0$. Comme expliqué dans le chapitre 5, les deux approches retenues permettent de ne redémarrer que le processeur en panne. Dans ce cas le coût avec une restauration est majoré par $T_p^f < T_p + O(T_\infty)$. Néanmoins, ce processeur peut exécuter une tâche sur le chemin critique du programme : le coût de restauration de ce processeur est alors $O(T_\infty)$.

En première approximation, dans le cas des programmes très parallèles avec $T_\infty \ll T_p$, le temps de reprise d'un des processeurs peut être négligé face au temps d'exécution. Dans ce cas, les surcoûts liés à l'utilisation des deux approches sont donnés par les résultats de la section précédente : seul le surcoût à l'exécution importe.

Si ce surcoût est important, alors dans le cas de l'approche basée sur la journalisation des événements de construction du graphe, le temps de ré-exécution dû à l'interprétation des événements jusqu'au point de la panne est lié au nombre de tâches non exécutées : il convient alors d'augmenter le grain de calcul en diminuant le nombre de tâches mais en perdant en degré de parallélisme. Dans l'approche basée sur une sauvegarde induite par le vol de travail, le temps nécessaire à revenir à la date de la panne est directement corrélé à la période.

Bibliographie

[1] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. Flowvr : a middleware for large scale virtual reality applications. In *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.

[2] K. Anstreicher, N. Brixius, J. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids, 2000.

[3] A. Avizienis. Fault-tolerant systems. *IEEE Trans. Computers*, 25(12) :1304–1312, 1976.

[4] A. Avizienis. Toward systematic design of fault-tolerant systems. *Computer*, 30(4) :51–58, 1997.

[5] A. Avizienis, J-C. Laprie, and B. Randell. Dependability and its threats - a taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, 2004.

[6] J. Baldeschwieler, R. Blumofe, and E. Brewer. Atlas : An infrastructure for global computing. In *Proceedis of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[7] R. Baldoni. A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, page 68. IEEE Computer Society, 1997.

[8] J. F. Bartlett. A non stop kernel. *the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, 1981.

[9] F. Baude, D. Caromel, C. Delbé, and L. Henrio. A hybrid message logging-cic protocol for constrained checkpointability. In *Proceedings of EuroPar2005*, LNCS, pages 644–653, Lisbon, Portugal, August-September 2005. Springer.

[10] S. Ben Hassen and H.E. Bal. Integrating task and data parallelism using shared objects. In *International Conference on Supercomputing*, pages 317–324, 1996.

[11] M. Bertier, O. Marin, and P. Sens. Performance analysis of hierarchical failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, San-Francisco (USA), June 2003. IEEE Society Press.

[12] X. Besseron, S. Jafar, T. Gautier, and J.-L. Roch. Cck : An improved coordinated checkpoint/rollback protocol for dataflow applications in kaapi. In *ICTTA'06 IEEE Conference on Information&Communication Technologies : from Theory to Applications*, Damascus, Syria, april 2006.

[13] B. Bhargava and S. R. Lian. Independent checkpointing and concurrent rollback for recovery–an optimistic approach. In *In Proceedings of the Symposium on Reliable Distributed Systems*, pages 3–12, 1988.

[14] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico.*, pages 356–368, November 1994.

[15] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk : An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1) :55–69, 1996.

[16] R.D. Blumofe and C.E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 1(27) :202–229, 1997.

[17] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. pages 133–147, 1997.

[18] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under unix. *ACM Trans. Comput. Syst.*, 7(1) :1–24, 1989.

[19] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. Mpich-v : Toward a scalable fault tolerant mpi for volatile nodes. In *SuperComputing*, Baltimore, USA, 2002.

[20] A. Bouteiller, F. Cappello, T. Hérault, P. Lemarinier, G. Krawezik, and F. Magniette. Mpich-v2 : a fault tolerant mpi for volatile nodes based on the pessimistic sender based message logging. In *SuperComputing*, Phoenix, USA, 2003.

[21] A. Bouteiller, P. Lemarinier, T. Hérault, G. Krawezik, and F. Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant mpi. In *In proceedings of The 2004 IEEE International Conference on Cluster Computing*, San Diego, USA, 2004.

[22] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. In *In proceedings of The 2003 IEEE International Conference on Cluster Computing*, Honk Hong,China, 2003.

[23] R. K. Brunner, J. C. Phillips, and L. V. Kale. Scalable molecular dynamics for large biomolecular systems. In *Supercomputing '00 : Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 45, Washington, DC, USA, 2000. IEEE Computer Society.

[24] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. pages 199–216, 1993.

[25] J. Buisson, F. André, and J-L. Pazat. Dynamic adaptation for grid computing. In *EGC*, pages 538–547, 2005.

[26] N. Capit, G. Da Costa, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. Expériences autour d'une nouvelle approche de conception d'un gestionnaire de travaux pour gr appe. In *Actes de CFSE 2003*, 2003.

[27] N. Capit, L. Desbat, L. Eyrauld, and O. Richard. Ciment grid : a grid facility for large scale parametric computing. In *Workshop PMAA*, October 2004.

[28] N. Capit, nd Y. Georgiou G. Da Costa, G. Huard, C. Marti n, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, 2005.

[29] A. Carissimi and M. Pasin. Athapascan : An experience on mixing mpi communications and threads. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 137–144, London, UK, 1998. Springer-Verlag.

[30] G. Cavalheiro. *Athapascan 1 : Interface genérique pour l'ordonnancement dans un environnement d'exécution parallèle*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, November 1999.

[31] CERT Coordination Center. Cert/cc statistics 1988-2004. Technical report, 2004.

[32] S. Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[33] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, 1996.

[34] K. M. Chandy and L. Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[35] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self adapting software for numerical linear algebra and lapack for clusters, 2003.

[36] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Building fault survivable mpi programs with ftmpi using diskless checkpointing. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. ACM Press, 2005.

[37] D. Conan. *Tolérance aux fautes par recouvrement arrière dans les systèmes informatiques répartis*. Phd thèse, informatique, Université PARIS 6, 1996.

[38] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems (3rd edtion) : Concepts and Design*. Addison-Wesley, 2001.

[39] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic broadcast : From simple message diffusion to byzantine agreement. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, Ann Arbor, MI, USA, 1985. IEEE Computer Society Press.

[40] J. DeVale. *Traditional Reliability. 18-849b Dependable Embedded Systems*. Spring, 1998.

[41] A. Djerrah, S. Jafar, V-D. Cung, and H. Peter. Solving qap on cluster with a bound of reformulation linearization techniques. In *In the 17th IMACS World Congress Scientific Computation, Applied Mathematics and Simulation IMACS2005*, Paris, France, July 2005.

[42] J. Dongarra and V. Eijkhout. Self-adapting numerical software for next generation applications. Technical report, Innovative Computing Laboratory, University of Tennessee, 2002.

[43] M. Doreille. *Athapascan 1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. Thèse de doctorat en mathématiques appliquées, Institut National Polytechnique de Grenoble, France, December 1999.

[44] E. N. Elnozahy. *Manetho : Fault-Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*. Phd thesis, Rice University, October 1993.

[45] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3) :375–408, 2002.

[46] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, oct 1996.

[47] R. Finkel and U. Manber. Diba distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.*, 9(2) :235–256, 1987.

[48] High Performance Fortran Forum. High performance fortran language specification, version 2.0, Janvier 1997.

[49] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1 : On-line building data flow graph in a parallel language. In IEEE, editor, *PACT'98*, pages 88–95, Paris, France, October 1998.

[50] François Galilée. *Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle*. Thèse de doctorat en informatique, INPG, septembre 1999.

[51] L. Gao and G. Malewicz. Internet computing of tasks with dependencies using unreliable workers. In *OPODIS*, pages 443–458, Grenoble, France, December 2004.

[52] T. Gautier and H.R. Hamidi. Re-scheduling invocations of services on rpc-based grid. *International Journal Computer Languages, Systems and Structures (CLSS)*, 2006. à paraître.

[53] T. Gautier, R. Revire, and Roch. Athapascan : Api for asynchronous parallel programming. Technical Report RR-0276, APACHE, INRIA Rhône-Alpes, February 2003.

[54] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM : Parallel Virtual Machine, A Users' Guide and Tutorial for Network Parallel Computing*. USA, November 1994.

[55] C. Germain and N. Playez. Result checking in global computing systems. In ACM, editor, *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS 03)*, San Francisco, California, 23–26 Juin 2003.

[56] J-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *HPDC*, pages 43–50, 2000.

[57] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6) :789–828, 1996.

[58] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4) :209–223, 1999.

[59] S. Jafar, T. Gautier, A. Krings, and J-L. Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In *Proceedings of EuroPar'05*, Lisboa, Portugal, August 2005.

[60] S. Jafar, T. Gautier, and J-L. Roch. Modèle de coût algorithmique intégrant des mécanismes de tolérance aux pannes et expérimentations. In *Actes des 16èmes rencontres francophones du parallélisme, RENPAR'16*, pages 125–136, Le Croisic, France, Avril 2005.

[61] S. Jafar, A. Krings, T. Gautier, and J-L. Roch. Theft-induced checkpointing for reconfigurable dataflow applications. In *Proceedings of the IEEE Electro/Information Technology Conference EIT2005*, Lincoln, Nebraska,U.S.A., May 2005. This paper received the IEEE EIT'05 Best Paper.

[62] S. Jafar, L. Pigeon, T. Gautier, and J.-L. Roch. Self-adaptation of parallel applications in heterogeneous and dynamic architectures. In *ICTTA'06 IEEE Conference on Information&Communication Technologies : from Theory to Applications*, Damascus, Syria, april 2006.

[63] S. Jafar and J.-L. Roch. Fault-tolerance for macro dataflow parallel computations on grid. In *ICTTA'04 IEEE Conference on Information&Communication Technologies : from Theory to Applications*, Damascus, Syria, april 2004.

[64] S. Jafar, S. Varrette, and J.-L. Roch. Using data-flow analysis for resilience and result checking in peer-to-peer computations. In *IEEE DEXA'2004 - Workshop GLOBE'04 : Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems*, Zaragoza, Spain, august 2004.

[65] KAAPI. Kernel for adaptative, asynchronous parallel interface, 2005. http ://www-id.imag.fr/Laboratoire/Membres/Gautier_Thierry/KAAPI/.

[66] L. Kal, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. Namd2 : greater scalability for parallel molecular dynamics. *J. Comput. Phys.*, 151(1) :283–312, 1999.

[67] L. V. Kale and S. Krishnan. *Parallel Programming using C++*, chapter Charm++ : Parallel Programming with Message-Driven Objects, pages 175–213. MIT Press, 1996.

[68] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.*, 13(1) :23–31, 1987.

[69] T.C. Koopmans and M.J. Beckmann. Assignment problems and the location of economic activities. *Econometrica*, 25, 1957.

[70] A. Krings, J-L. Roch, and S. Jafar. Certification of large distributed computations with task dependencies in hostile environments. In *Proceedings of the IEEE Electro/Information Technology Conference EIT2005*, Lincoln, Nebraska,U.S.A., May 2005.

[71] A. Krings, J-L. Roch, S. Jafar, and S. Varrette. A probabilistic approach for task and result certification of large-scale distributed applications in hostile environments. In *Proc. European Grid Conference (EGC2005), in LNCS 3470, Springer Verlag*, Amsterdam, Netherlands, February 2005.

[72] K. Marzullo L. Alvisi. Message logging : Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2) :149–159, 1998.

[73] K. Marzullo L. Alvisi. Message logging : Pessimistic, optimistic, causal and optimal. *TSE*, 24(2) :149–159, 1998. Transactions on Software Engineering.

[74] T.H. Lai and T.H. Yang. On distributed snapshots. *Information Processing Letters*, 25, May 1987.

[75] L. Lamport. Time, clocks, and the ordering of events in a distributed system, 1978.

[76] L. Lamport, 1987. http ://research.microsoft.com/users/lamport/pubs/distributed-system.txt.

[77] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3) :382–401, 1982.

[78] J-C. Laprie. *ARAGO 15 : Concepts de base de la tolérance aux fautes*. MASSON, Paris, 1994.

[79] J.C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability : Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[80] F. C. H. Lin and R. M. Keller. Distributed recovery in applicative systems. In *ICPP*, pages 405–412, 1986.

[81] F. C. H. Lin and R. M. Keller. Distributed recovery in applicative systems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 405–412, PA, USA, 1986.

[82] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification. Second Edition*. 1999.

[83] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[84] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report CS-TR-97-1346, Univ. Wisconsin, Madison, 1997.

[85] K. H. Randall M. Frigo, C. E. Leiserson. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223. ACM Press, 1998.

[86] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA '87 : Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[87] O. Marin, M. Bertier, and P. Sens. Darx - a framework for the fault-tolerant support of agent software. In *Proceedings of the 14th International Symposium on Software Reliability Engineering ISSRE'03*, pages 406 – 416, nov 2003.

[88] T. Mauto and C. Roucairol. A new exact alagorithm for the solution of quadratic assignment problems. *Disc. Appl. Math.*, 55 :281–293, 1994.

[89] K. R. Mazouni. Etude de l'invocation entre objets dupliqués dans un système tolérant aux fautes. Technical report, 1996.

[90] MOAIS. Inria project : Multi-programmation et ordonnancement sur ressources distribuées pour les applications interactives de simulation, 2005. http ://www.inria.fr/recherche/equipes_ur/moais.fr.html.

[91] J. Montagnat, V. Breton, and I. Magnin. Partitioning medical image databases for content-based queries on grid. *Methods Inf Med.*, 44(2) :154–60, 2005.

[92] R.H.B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2) :165–169, 1995.

[93] A. Nguyen-Tuong, A. S. Grimshaw, and M. Hyett. Exploiting data-flow for fault-tolerance in a wide-area parallel system. In *Proceedings 15 th Symposium on Reliable Distributed Systesm*, pages 2–11, 1996.

[94] N. Maillard P. Manneback J. L. Roch O. Beaumont, E.M Daoudi. Tradeoff to minimize extra-computations and stopping criterion tests for parallel iterative schemes. In *3rd International Workshop on Parallel Matrix Algorithms and Applications (PMAA04)*, CIRM, Marseille, France, 18–22 october 2004.

[95] J. S. Plank, H. Casanova, M. J. Beck, and J. Dongarra. Deploying fault tolerance and task migration with NetSolve. *Future Generation Computer Systems*, 15(5–6) :745–755, 1999. Elsevier.

[96] J.S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, 1983.

[97] D. Powell. Failure mode assumptions and assumption coverage. In *Proc. Fault-Tolerant Computing FTCS-22, Boston, MA, USA*, page 1992.

[98] D. Powell. Failure mode assumption coverage. *In 22th IEEE Sumposium on Fault tolerant Computing*, 1992.

[99] D. Powell, P. Verissimo, G. Bonn, F. Waeselynck, and D. Seaton. The delta-4 approach to dependability in open distributed computing systems. In *Proc. Fault-Tolerant Computing FTCS-18, Tokyo (J)*, page 1988.

[100] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, 1975.

[101] R. Revire. *Ordonnancement de graphe dynamique de tâches sur architecture de grande taille. Régulation par dégénération séquentielle et distribuée*. Thèse de doctorat en informatique, INPG, septembre 2004.

[102] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3) :483–545, 1 May 1998.

[103] M.C. Rinard and M.S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Programming Languages and Systems*, 20(3) :483–545, 1998.

[104] D. L. Russell. State restoration in systems of communicating processes. *IEEE Trans. Software Eng.*, 6(2) :183–194, 1980.

[105] L. F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Gener. Comput. Syst.*, 18(4) :561–572, 2002.

[106] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3) :149–157, April 1997.

[107] A. Schiper. Failure detection vs. group membership in fault-tolerant distributed systems : Hidden trade-offs. In *Proceedings PAPM-PROBMIV 2002*, LNCS 2399, pages 1–15, Copenhagen, Denmark, July 2002. Springer Verlag. Invited talk.

[108] R. D. Schlichting and F. B. Schneider. Fail-stop processors : An approach to designing fault-tolerant computing systems. *Computer Systems*, 1(3) :222–238, 1983.

[109] F. B. Schneider. Byzantine generals in action : implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2) :145–154, 1984.

[110] F. B. Schneider. Implementing fault-tolerant services using the state machine approach : a tutorial. *ACM Comput. Surv.*, 22(4) :299–319, 1990.

[111] N. Sergent, X. Défago, and A. Schiper. Impact of a failure detection mechanism on the performance of consensus. In *Proc. IEEE Pacific Rim Symp. on Dependable Computing (PRDC)*, Seoul, Korea, December 2001.

[112] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems : Design and Evaluation, Second Edition*. Butterworth, December 1991.

[113] G. Stellner. CoCheck : Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[114] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3) :204–226, 1985.

[115] V. Strumpen. Compiler technology for portable checkpoints. Technical Report MA-02139, MIT Laboratory for Computer Science, Cambridge, 1998.

[116] Y. Tamir and C. Sequin. Error recovery in multicomputers using global checkpoints. In *Proc. of 1984 International Conference on Parallel Processing*, pages 32–41, August 1984.

[117] A. S. Tanenbaum and M. Steen. *Distributed Systems : Principles and Paradigms*. Prentice Hall, 2002.

[118] M. Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *ArXiv Computer Science e-prints*, pages 1002–+, dec 2005.

[119] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley and Sons, New York, 2001.

[120] S. Vadhiyar, E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Proceedings of Super Computing 2000*, pages 46–56, Dallas, Texas, November 2000.

[121] S. S Vadhiyar and J. J Dongarra. Self adaptivity in grid computing. *Concurrency and Computation*, 17(24) :235–257, 2005.

[122] L.G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8) :103–111, 1990.

[123] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin : Simple and efficient java-based grid programming. *Scalable Computing : Practice and Experience*, 6(3) :19–32, September 2005.

[124] Y. Wang, P. Chung, I. Lin, and W.K.Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5) :546–554, May 1995.

[125] R. Clint Whaley and J. Dongarra. Self adapting linear algebra algorithms and software. In *SC'98 : High Performance Networking and Computing*, 1998.

[126] M. Wiesmann, F. Pedonne, and A. Schipper. A systematic classification of replited database protocols based on atomic broadcast. In *Proceedings of the 3th European Research Seminar on Advances in Distributed Systems (ERSADS99)*, pages 351–360, 1999.

[127] G. Wrzesinska, R. V. Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Fault-tolerant scheduling of fine-grained tasks in grid environments. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(1), Feb. 2006.

[128] Gosia Wrzesinska, Rob van Nieuwpoort, Jason Maassen, and Henri E. Bal. Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid.

In *Proc. of 19th International Parallel and Distributed Processing Symposium*, Denver, CO, USA, April 2005.

[129] G. Zheng, L. Shi, and L. V. Kalé. Ftc-charm++ : An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September 2004.

Résumé. **Programmation des systèmes parallèles distribués : mécanismes de tolérance, résilience et adaptabilité :**

Les grilles et les grappes sont des architectures de plus en plus utilisées dans le domaine du calcul scientifique distribué. Le nombre important de constituants hétérogènes (processeurs, mémoire, interconnexion) dans ces architectures dynamiques font que le risque de défaillance est très important. Compte tenu de la durée considérable de l'exécution d'une application parallèle distribuée, ce risque de défaillance doit être contrôlé par l'utilisation de technique de tolérance aux pannes. Dans ce travail, la représentation de l'état de l'exécution d'un programme parallèle est un graphe, dynamique, de flot de données construit à l'exécution. Cette description du parallélisme est indépendante du nombre de ressources et donc exploitée pour résoudre les problèmes liés à la dynamicité des plateformes considérées. La définition de formats portables pour la représentation des noeuds du graphe résout les problèmes d'hétérogénéité. La sauvegarde du graphe de flot de données d'une application durant son exécution sur une plateforme, constitue des points de reprise pour cette application. Par la suite, une reprise est possible sur un autre type ou nombre de processus. Deux méthodes de sauvegarde / reprise, avec une analyse formelle de leurs complexités, sont présentées : SEL (*Systematic Event Logging*) et TIC (*Theft-Induced Checkpointing*). Des mesures expérimentales d'un prototype sur des applications caractéristiques montrent que le surcoût à l'exécution peut être amorti, permettant d'envisager des exécutions tolérantes aux pannes qui passent à l'échelle.
Mots-clés. Tolérance aux pannes, Systèmes répartis, Grille, Point de reprise, Recouvremment.

Abstract. **Parallel and distributed systems programming : Fault-Tolerance, resilience and adaptability :**

Grid and cluster architectures are gaining in popularity for scientific computing applications. The distributed computations, as well as their underlying infrastructure consisting of a large number of computers, storage and networking devices, pose challenges in overcoming the effects of node failures. This work presents a new checkpoint/recovery method for dataflow computations using work-stealing in heterogeneous environments as found in grid or cluster computing. Basing the state of the computation on a dynamic macro dataflow graph, it is shown that the mechanisms provide effective checkpointing for multithreaded applications in heterogeneous environments. Two methods are presented, i.e. *Systematic Event Logging* (SEL) and *Theft-Induced Checkpointing* (TIC), which are efficient and extremely flexible under the system-state model, allowing for recovery on different platforms under different number of processors. A formal analysis of the overhead induced by both methods is presented, followed by an experimental evaluation in a large platform. It is shown that both methods have very small overhead and that trade-offs between checkpointing and recovery cost can be controlled.
Keywords. Fault-Tolerance, Distributed systems, Grid computing, Checkpoint, Recovery.