**RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants**
**Le Croisic, France, 5 au 8 avril 2005**

# Modèle de coût algorithmique intégrant des mécanismes de tolérance aux pannes et expérimentations

Samir JAFAR, Thierry GAUTIER et Jean-Louis ROCH

Projet APACHE, Laboratoire Informatique et Distribution ID-IMAG,
51 av Jean Kuntzmann, 38330 Montbonnot Saint Martin, France.
Samir.Jafar@imag.fr, Thierry.Gautier@inrialples.fr, Jean-Louis.Roch@imag.fr

---

**Résumé**
Les grilles et les clusters sont des architectures de plus en plus utilisées dans le domaine du calcul scientifique distribué. Le nombre important de constituants (processeurs, mémoire, interconnexion) dans ces architectures font que le risque de défaillance est très important. Compte-tenu de la durée considérable de l'exécution d'une application distribuée, ce risque de défaillance doit être contrôlé par l'utilisation de technique de tolérance aux pannes. Dans cet article, nous présentons deux mécanismes de tolérance aux pannes basés sur une sauvegarde de l'état du *futur* de l'exécution représenté par un graphe de flot de données. Nous présentons leurs modèles de coût algorithmique intégrant le temps nécessaire pour la sauvegarde de l'état des processus. Nous montrons que pour la classe des programmes considérée et les mécanismes de tolérance aux pannes, les accélérations asymptotiques sont linéaires en fonction du nombre de processeurs. Un prototype existe et des expérimentations montrent que le surcoût à l'exécution peut être amorti, permettant d'envisager des exécutions tolérantes aux pannes qui passent à l'échelle. Des comparaisons expérimentales sur une grappe d'environ 200 processeurs complètent les analyses théoriques.

**Mots-clés :** Tolérance aux pannes. Graphe de flot de données. Application à très grande échelle. Grille.

---

## 1. Introduction

L'utilisation d'un grand nombre de ressources de calcul hétérogènes augmente fortement le risque d'apparition d'une panne au cours de l'exécution d'une application distribuée [13]. Par exemple dans [2], les auteurs relatent leur expérience sur l'exécution d'une application durant 15 jours sur une grille de calcul d'environ 650 processeurs en moyenne (jusqu'à un maximum de 1000 processeurs). Durant cette exécution, la grille s'est arrêtée 5 fois pour des raisons de maintenance et de pannes.
De nombreuses études ont proposées des protocoles [11] qui permettent de reprendre ou continuer une exécution après une défaillance d'un processus communiquant par message. Outre les approches basées sur la *duplication* [22] qui ne permettent que de tolérer un nombre fixé de pannes, tous ces protocoles se basent sur, d'une part, la *sauvegarde* d'un état des processus du système, et, d'autre part, sur la construction d'un état global cohérent lors de l'étape de *restauration* du contexte d'un processus. Les approches basées sur la *journalisation des messages échangés* [11, 7] s'appuient sur le fait qu'un processus peut être modélisé par une séquence

d'intervalles d'états [23] chacun débutant par un événement dont l'enregistrement permet la reconstruction de l'état du processus.
Quatre critères fondamentaux permettent alors de comparer ces différents protocoles de sauvegarde et reprise :

**état faible ou fort** : suivant que l'état sauvegardé consiste en seulement des données en mémoire et du code (*état faible*), ou bien contienne aussi l'état d'exécution des processus en cours (*état fort*).

**coordination** : si les processus se coordonnent afin de construire un état global cohérent au moment de la sauvegarde, l'approche est appelée *sauvegarde coordonnée* [19, 11], dans le cas contraire celle-ci est dite *sauvegarde non coordonnée* [11] et requièrt la reconstruction, au moment de la reprise, d'un état cohérent.

**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*.

**restauration globale ou locale** : si la restauration après une défaillance nécessite la construction d'un état cohérent global pour redémarrer l'application, le protocole est dit à *restauration globale*. Dans le cas contraire, le protocole est dit à *restauration locale*, et seule une connaissance de l'état du processus avant défaillance, voir des processus dans son voisinage local, permet le redémarrage de l'application.

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 de sauvegarde / reprise. La seconde concerne le «**passage à l'échelle**» de ces protocoles sur plusieurs (centaines de) milliers de processeurs.
L'objectif de cet article est donc la construction d'applications distribuées qui soient performantes et tolérantes aux pannes sur des grilles de calcul. Nous traitons les pannes franches des processus [16] ainsi les déconnexions des nœuds. Nous supposons posséder un détecteur de défaillance [9], capable de détecter les pannes et d'identifier le composant défaillant dans le système.
Nous considérons cette problématique dans le contexte du calcul multiflots (en anglais *multithreaded computation*) pour lesquel il existe des algorithmes d'ordonnancement efficaces en théorie [12] comme en pratique [5] et sur un grand nombre de processeurs. Nous étendons un modèle de coût algorithmique afin d'ajouter les coûts d'un protocole pour la sauvegarde / retauration. Nous montrons que pour deux mécanismes de tolérance aux pannes considérés ici, la complexité en temps et en mémoire de toute exécution d'une application est du même ordre que sans ces mécanismes. Nous montrons de plus comment amortir ces surcoûts par une adaptation algorithmique du grain de calcul ou par une implantation fine utilisant le «*principe du travail d'abord*» [1] [5]. Nous illustrons notre présentation par un prototype implantant deux protocoles de sauvegarde / reprise s'appuyant sur notre intergiciel KAAPI et son modèle d'exécution basé sur un algorithme de vol de travail distribué.

[1] En anglais *work first principle*.

La section suivante présente le positionnement de notre travail vis-à-vis de l'état de l'art. La section 3 présente KAAPI, la section 3.2 son modèle d'exécution et son modèle de coût. Dans la section 4, nous présentons l'implantation de deux protocoles de sauvegarde / restauration au niveau de l'ordonnanceur de KAAPI. La section 5 présente le modèle de coût tenant compte de protocoles de sauvegarde / restauration. Enfin, la section 6 présente les résultats expérimentaux sur une application d'optimisation combinatoire.

## 2. État de l'art et positionnement

La tolérance aux pannes s'inscrit dans le contexte plus large de la sûreté de fonctionnement. La sûreté de fonctionnement d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance dans le service qu'il délivre. La tolérance aux pannes est la mise en œuvre de cette propriété par l'assemblage de deux techniques : le *traitement de la faute* qui vise à éviter qu'une faute survenue ne se reproduise et le *traitement d'erreur* qui vise à éliminer une erreur avant qu'elle ne produise une *défaillance* [20]. La tolérane aux pannes est toujours traitée par l'emploi d'une **redondance** des calculs (traitements multiples) ou des moyens de stockage (duplication de composants).

Plusieurs projets de recherche ont étudié l'intégration de la tolérance aux pannes par des mécanismes de **sauvegarde/restauration** d'état dans le cadre d'environnements parallèles. Condor [21] fournit une bibliothèque qui permet de réaliser une sauvegarde de l'état d'un processus. Cette bibliothèque a été conçue pour des applications sans dépendances de données entre processus [10]. Ses principales limitations sont que Condor ne peut pas sauvegarder l'état d'un processus contenant plusieurs threads et que l'état sauvegardé n'est pas hétérogène.

MPICH-V1 [7] est basé sur la sauvegarde non coordonnée avec journalisation des messages. Afin de pouvoir redémarrer uniquement les processus défaillants, MPICH-V1 utilise le concept du canal mémoire ; tous les messages échangés passent par un canal mémoire supposé fiable. Cette technique implique qu'il n'est pas possible d'avoir des communications directes entre les processus. Pour éviter l'utilisation du canal mémoire MPICH-V2 [8] journalise toutes les communications sortantes. Les deux versions de MPICH-V utilisent la bibliothèque de checkpoints fournit par Condor [21] et donc ne supportent pas l'hétérogénéité ni le multithreading.

Mentat [1] est un environnement de programmation parallèle basé sur la représentation d'un programme par un graphe de flot de données. En se basant sur ce graphe, Mentat supporte la tolérance aux pannes par duplication des tâches sur toutes les ressources de calcul. Malgré des stratégies de duplication évoluées, le surcoût de gestion de la tolérance aux pannes reste important.

ProActive [3] est une bibliothèque Java pour le calcul distribué basée sur la notion d'*objets actifs*. ProActive propose un protocole de tolérance aux pannes par sauvergarde / restauration induite par les messages. L'état sauvegardé est constitué de chacun des objets actifs à des états bien définis. ProActive implante une *restauration globale* : la restauration après panne nécessite l'arrêt et la reprise de tous les processus.

Les outils de sauvegarde proposés dans les systèmes existants ont été réalisés pour des utilisations spécifiques dont l'évaluation reste seulement expérimentale [7, 8, 25]. Ils ne sont généralement pas réutilisables. De plus, l'état sauvegardé n'est pas hétérogène [21, 7, 8], si ce n'est parce que le langage est portable, comme Java [3], ou parce que l'outil s'insère dans le processus de compilation, tel Porch [25]. Ajoutons que tous ces systèmes nécessitent un nouveau processus en remplacement de la défaillance d'un processus.

## 3. KAAPI

KAAPI signifie *Kernel for Adaptative and Asynchronous Parallel Interface*. Il s'agit d'un intergiciel qui permet la programmation d'une application parallèle et distribuée. KAAPI est un moteur exécutif pouvant être utilisé comme machine cible pour des langages de plus haut niveau, comme Athapascan [15].

### 3.1. Présentation générale

Une application de KAAPI décrit à l'exécution un graphe de flot de données qui représente un enchaînement de tâches qui se synchronisent par des accès à des données partagées. La description du parallélisme est indépendante de l'architecture distribuée sous-jacente : un algorithme d'ordonnancement dynamique décide de la machine cible pour l'exécution des tâches et le stockage des données.
L'implantation de KAAPI repose sur une architecture de machine virtuelle constituée par un ensemble dynamique de processus qui communiquent par échange de message actif. Une ressource de calcul est représentée dans KAAPI par la notion de processeur, appelé *K-processeur* : un *K-processeur* possède plusieurs flots d'exécution, appelés *K-threads*, dont au plus un est en cours d'exécution. Un processus peut possèder plusieurs *K-processeurs* qui peuvent alors tirer parti de l'espace d'adressage en commun et communiquer par partage de la mémoire. Les notions de *K-processeur* et *K-thread* doivent être spécialisées en fonction de l'application et du système d'exploitation sous-jacent[2]. Dans ATHAPASCAN, un *K-thread* est implanté par un thread POSIX de niveau kernel, le *K-processeur* gérant l'activité des plusieurs threads POSIX afin de respecter les contraintes imposés par KAAPI.

### 3.2. Modèles de programmation et d'exécution

Un programme KAAPI commence toujours par l'exécution d'une tâche particulière, appelée tâche principale, dans le contexte d'exécution du *K-thread* actif sur l'un des *K-processeurs*. Une tâche est un appel de fonction, c'est-à-dire une fonction ainsi que ses paramètres effectifs. Le mode de passage des paramètres se fait par valeur (copie) ou par référence en précisant la donnée globale ainsi que le mode d'accès (lecture, écriture ou modification) que fait la tâche à la donnée. Une tâche (mère) peut créer des tâches (filles) qui communiquent par des données partagées qui sont soit passées en paramètres effectifs, soit déclarées dans la tâche mère. Une tâche devient prête lorsque tous ses paramètres sont prêts, c'est-à-dire lorsque tous les paramètres effectifs passés en lecture et par référence sont produits par une tâche précédemment créée [24].
Le modèle de programme ainsi considéré est du type *calcul multiflots* [6, 12, 15] comme dans Athapascan [15] ou Cilk[3] [6]. Dans ATHAPASCAN, à la différence de KAAPI, le graphe de flot de données entre les tâches est calculé au cours de l'exécution grâce à une technique d'interprétation [15]. La représentation à l'exécution des programmes est un *graphe de flot de données* : un graphe orienté, sans cycle, biparti composé de nœuds «données» et de nœuds «tâches».
Cette classe de programme peut être efficacement exécutée sur $p$ processeurs. L'algorithme d'ordonnancement est basé sur une technique distribuée de vol de travail [12, 5]. Le principe est le suivant : si un *K-processeur* est inactif, c'est-à-dire qu'il n'a aucun *K-thread* éligible à l'exécution, alors il devient un *voleur*. Un voleur choisit un *K-processeur* victime et tente de lui voler du travail (une tâche) dans l'un de ses *K-thread*. La tâche volée est marquée comme étant *bloquée*, et une copie est faite dans le contexte d'exécution du *K-thread* voleur. A la fin de l'exécution de la tâche copiée, les données produites sont écrites dans la mémoire globale et la tâche volée est marquée

[2] En particulier de la bibliothèque de thread disponible sur le système cible.
[3] En pratique Cilk ne considère que les programmes *fully strict multithreaded computation*.

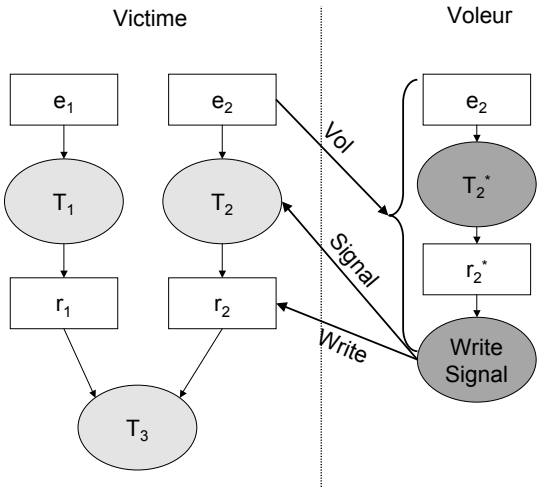



Figure 1: Illustration du vol entre deux $K$-processeurs sur le graphe de flot de données. Le $K$-processeur voleur copie la tâche $T_2$ avec ses paramètres d'entrées ($e_2$) et crée une tâche $WriteSignal$ qui sera chargée d'écrire son paramètre ($r_2^\star$) dans la mémoire occupée par $r_2$ et de signaler à la tâche $T_2$ que le vol est terminé.

*exécutée*. L'opération de vol et la production des données en résultat sont les deux seules communications entre deux *K-processeurs*. Cette communication se traduit par un message actif dans le cas où les *K-processeurs* résident dans deux processus distincts. La figure 1 illustre l'opération de vol entre deux $K$-processeurs.

Notons par $T_1$ le temps séquentiel, $T_\infty$ le temps du chemin critique, et par $S_1$ l'espace mémoire requis par une exécution séquentielle [15, 6]. Dans le cadre restreint des grappes homogènes et en négligeant les communications, nous obtenons le théorème suivant.

**Théorème 1** *Le temps d'exécution $T_p$ d'un programme* KAAPI *avec l'algorithme de vol de travail sur $p$ processeurs homogène est [15, 6] : $T_p \leq c_1 T_1/p + c_\infty T_\infty$. Le nombre de vols par processeur est au plus $O(T\infty)$. L'espace mémoire sur $p$ processeurs est [6] : $S_p \leq pS_1$.*

Les constantes $c_1$ et $c_\infty$ 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 $c_\infty$. Dans l'implantation de Cilk [5], les auteurs montrent qu'il est possible de réaliser une implantation fine telle que pour de nombreuses applications $c_1 \simeq 1$. Cette constante est très importante car elle mesure le surcoût ajouté par l'implantation pour la gestion du parallélisme ou, comme nous le verrons ci-dessous, pour la gestion de la tolérance aux pannes, en plus du travail de l'application, *i.e.* le nombre d'opérations de l'application. Le principe du *travail d'abord* 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.*

## 4. Intégration d'une fonctionnalité pour la tolérance aux pannes

Les sections suivantes présentent les deux protocoles pour la tolérance aux pannes que nous avons intégrés dans KAAPI. Le premier se base sur la journalisation des événements de changement d'état. Le second se base sur une sauvegarde périodique et induite par les communications.

### 4.1. Choix de l'état à sauvegarder

Les deux protocoles sont basés sur le graphe de flot de données comme représentation naturelle d'une exécution à la fois pour la représentation du parallélisme et pour la tolérance aux pannes [1]. Ce graphe est une représentation de l'ensemble du calcul à exécuter. Il est dynamique et évolue au cours de l'exécution.

Dans ces deux protocoles, nous avons implanté la sauvegarde d'un **état faible**, l'état d'exécution des $k$-threads de l'intergiciel. Les points où la sauvegarde est possible sont avant ou après l'exécution d'une tâche de l'application. L'exécution d'une tâche n'est pas sauvegardable. Cette solution permet la sauvegarde portable sur une architecture **hétérogène** comme n'importe quelle structure de données.

Le graphe de flot de données sauvegardé contient uniquement le futur de l'exécution : les tâches à exécuter ainsi que les données nécessaires. Certaines données temporaires ou non nécessaires au futur de l'exécution, bien que pouvant résider en mémoire, ne sont pas sauvegardées. En ce sens, l'utilisation du graphe de flot de données permet de réduire la taille mémoire des fichiers de sauvegarde.

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.

### 4.2. Protocoles implantés

Les deux protocoles retenus visent à limiter le surcoût dû à la sauvegarde dans un fichier (sur un support supposé persistant) de l'état des processus KAAPI. Le premier protocole s'inspire des protocoles de sauvegarde basés sur la journalisation des messages entre processus qui communiquent [11]. Seuls les événements de changement d'état sont sauvegardés, c'est-à-dire les ajouts et suppressions de nœuds dans le graphe de flot de données ainsi que les productions de valeurs. Le second protocole se base sur la sauvegarde non coordonnée induite par les communications; en effet, d'après le théorème 1, le nombre de vols qui provoquent les communications entre processus est faible dans le cas des applications très parallèles où $T_\infty \ll T_1$.

#### 4.2.1. Journalisation des événements de construction du graphe

Les protocoles basés sur la *journalisation des messages échangés* [11, 7] s'appuient sur le fait qu'un processus peut être modélisé par une séquence d'intervalles d'états, chacun débutant par un événement non déterministe [23]. Sous l'hypothèse que chaque événement non déterministe peut être identifié à un événement déterministe, l'enregistrement de ces derniers permet de rejouer les changements d'état des processus.

Appliqué à KAAPI où l'état à sauvegarder d'un processus est représenté par le sous graphe de flot de données de l'application associé à ce processus, ce protocole consiste à sauvegarder les événements de changement d'état du graphe de flot de données. De cette manière, l'état du graphe est sauvegardé de manière incrémentale et un mécanisme de ré-interprétation permet une reprise du calcul [17]. L'avantage de cette méthode est de permettre de rejouer une seule tâche, ce qui est intéressant pour des applications nécessitant la certification des calculs [18]. L'implantation est la suivante. Chaque nœud du graphe de flot de données est associé à un identificateur unique durant l'exécution du programme parallèle et est enregistré sur un support stable. Toute modification en cours d'exécution sur l'un des nœuds du graphe est enregistrée.

#### 4.2.2. Sauvegarde périodique

Le second mécanisme se base une sauvegarde non coordonnée induite par les communications [11] du graphe de flot de données de l'application. Des sauvegardes du graphe sont enregistrés périodiquement. De plus, l'état du graphe est enregistré lors des opérations de vol qui génèrent les seules communications de l'application. Ces sauvegardes sont appelées *sauvegardes forcées*.

L'avantage de ce protocole *non coordonnée* est que son surcoût à l'exécution peut être facilement amorti, comme nous le montrerons dans la section suivante. L'implantation dans KAAPI applique le principe ci-dessus pour la sauvegarde des $K$-processeurs au sein d'un même processus. De cette manière, chaque processus est sauvegardé de manière incrémentale par la sauvegarde de chacun de ses $K$-processeurs. Des sauvegardes de l'état des $K$-processeurs sont ajoutées lors des opérations de vol de travail. De cette manière, une sauvegarde d'un $K$-processeur peut être recouverte par le calcul effectué par un autre $K$-processeur.

## 5. Evaluation des performances

Dans cette section, 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 comme décrit dans la section 3.2.

L'architecture considérée est supposée homogène comme précisée dans les hypothèses du théorème 1. Le temps d'exécution d'une application sur $p$ processeurs est donc $T_p \leq c_1 T_1/p + c_\infty T_\infty$. Nous noterons par $\sigma$ le nombre de tâches créées au cour 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 $\tau_s$.

### 5.1. 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. La section 5.2 présente le modèle de coût d'une exécution avec une panne.

#### 5.1.1. Modèle de coût avec la journalisation des événements de construction du graphe

Chaque modification dans le graphe de flot de données induit une écriture sur le support stable : chaque création et suppression de tâches, 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*, 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 :

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

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

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

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 $\sigma$ ; au niveau de l'implantation, il est important d'utiliser un réseau rapide afin de diminuer $\tau_s$ 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 réelles de $\tau_s$ et du nombre de tâches crées.

#### 5.1.2. Modèle de coût avec la sauvegarde périodique

Dans le cadre de la sauvegarde périodique, chaque $K$-thread enregistre son graphe de flot de données sur le support stable toutes les $k$ 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 $K$-thread est $\tilde{\tau_s} = O(T_\infty \tau_s)$.
Le nombre de $K$-threads actifs par processeur est au plus 1. Le nombre de sauvegardes réalisées par un $K$-thread durant son exécution est au plus $\frac{T_p}{k}$. Le nombre de sauvegardes forcées dues aux opérations de vols est au plus $O(T_\infty)$ par $K$-processeur. En conséquence, le nombre total de sauvegardes $N$ réalisées par chaque $K$ threads est :

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

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

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

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 [4].

### 5.2. 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 précédent 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 les sections 4.2.1 et 4.2.2, 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 les 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

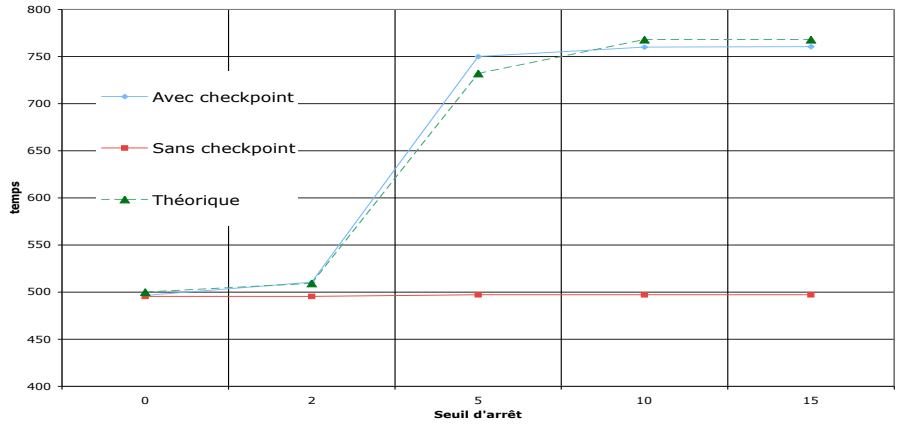
de parallélisme. Dans l'approche basée sur une sauvegarde périodique, le temps nécessaire à revenir à la date de la panne est directement corrélé à la période.

## 6. Expérimentation avec «DOC-G»

DOC-G (Défi en optimisation Combinatoire sur Grille) est une ACI GRID (partenaires PRiSM, ID-IMAG, LIFO, coordinateur: V.-D. Cung) dont l'objectif est d'exploiter des architectures de type grille de calcul pour la résolution de problèmes d'obtimisation combinatoire de type *Branch-and Bound*. Dans le cadre de ce projet, une application de $QAP$ (Quadratic Assignment Problem) a été parallélée avec KAAPI.
Les objectifs de ces expériences sont d'illustrer les modèles de coût algorithmique de la section précédente en utilisant un grand nombre de processeurs. Les expériences ont été effectuées sur le iCluster2[4] de l'IMAG hébergé à l'INRIA Rhône-Alpes en utilisant plusieurs instances, «NUGENT», du QAP. L'architecture de KAAPI avec les protocoles de tolérance aux pannes se base sur NFS pour implanter le support persistant : chaque nœud de calcul voit le support persistant à travers un montage NFS. Le choix simple de NFS permet une évaluation des coûts d'exécution et les corréler aux modèles proposés. Ce premier prototype sera suivi d'un second possédant un support persistant distribué [14].

### 6.1. 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. La figure 2 compare les temps d'exécution




| $Seuil$ | $\sigma$ |
|---|---|
| 0 | 1 |
| 2 | 262 |
| 5 | 6481 |
| 10 | 7491 |

Figure 2: Nug17s sur 1 processeur

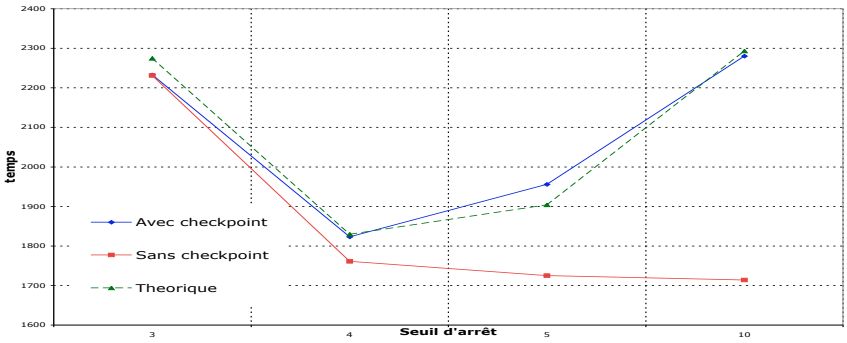
avec et sans protocole de tolérance en fonction du seuil d'arrêt de création du parallélisme de l'application. La courbe «théorique» représente le temps donné par notre modèle après estimation des paramètres ($\tau_s \simeq 7.16E^{-5}$, $\gamma^* \simeq 1.009 + \sigma\tau_s$). Pour un petit grain (seuil = 5 ou 10), le nombre de tâches est important et le surcoût dû au protocole represente plus de $50\%$ du temps d'exécution. En augmentant le volume de calcul en utilisant l'instance NUG18S du QAP, ce surcoût est d'environ $10\%$ pour environ $4000$ tâches pour un temps d'exécution de $2286s$. Ces résultats montrent la pertinence de notre modèle.

### 6.2. Influence du grain de l'application sur 25 noeuds

Les résultats précédents montrent que si le choix du grain peut réduire le surcoût dû au protocole de tolérance aux pannes, il réduit de même le degré de parallélisme de l'application. La

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

figure 3 présente les mesures expérimentales obtenues pour l'instance NUG22s sur 25 noeuds de la grappe. Dans la version sans protocole de tolérance aux pannes, le temps décroît lorsque l'on augmente le grain de parallélisme : de plus en plus de tâches sont créées jusqu'à permettre aux processeurs d'être quasiment toujours actifs. L'efficacité mesurée sur 25 processeurs avec un seuil d'arrêt du parallélisme de 10 est de $98\%$ contre $75\%$ pour un seuil de 3.




| $Seuil$ | $\sigma$ |
|---|---|
| 0 | 1 |
| 3 | 1678 |
| 4 | 21288 |
| 5 | 61967 |
| 10 | 221480 |

Figure 3: Nug22s sur 25 processeurs.

En revanche, lorsque le grain de parallélisme diminue, le temps d'exécution avec la version avec protocole de tolérance aux pannes augmente : cela est dû au paramètre en $O(\tau_s \sigma)$ de l'équation 1 linéaire en le nombre $\sigma$ de tâches.

### 6.3. Comparaison des deux protocoles

La figure 4 montre les différences de temps entre le protocole «systématique» et le protocole «périodique» pour deux périodes (une sauvegarde toute les secondes et toute les 20 secondes). Les mesures de la méthode périodique montre que le surcoût est très faible vis-à-vis de l'exécution sans sauvegarde (entre $0.6\%$ et $1\%$). L'écart entre les deux protocoles montre

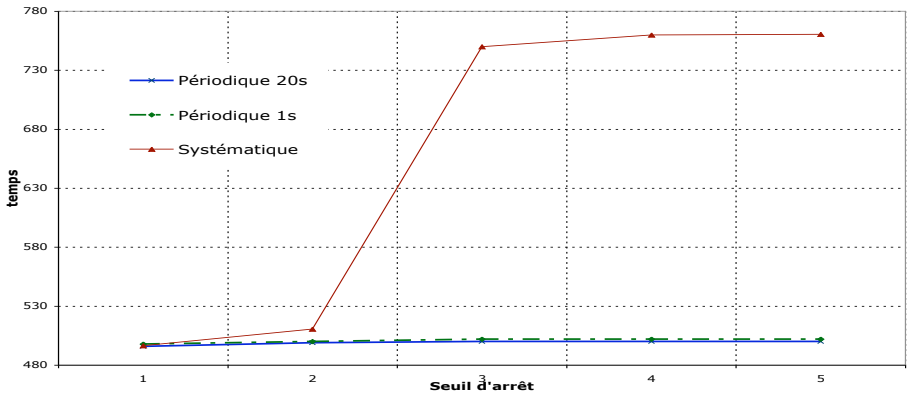



Figure 4: Comparaison des protocoles «systématique» et «périodique».

un surcoût important de l'approche systématique dû à la sauvegarde d'un grand ensemble d'événements. Réduire ce coût revient soit à augmenter le grain de l'application et donc perdre en degré de parallélisme et en efficacité ; soit à retarder les écritures sur le support stable en désynchronisant la sauvegarde effectives des événements, c'est-à-dire en utilisant une approche de type périodique.

### 6.4. Influence du nombre de processeurs

La figure 5 montre le gain en fonction du nombre $p$ de processeurs des exécutions sur l'instance NUG22S du QAP sans protocole de sauvegarde et avec le protocole systématique. La première

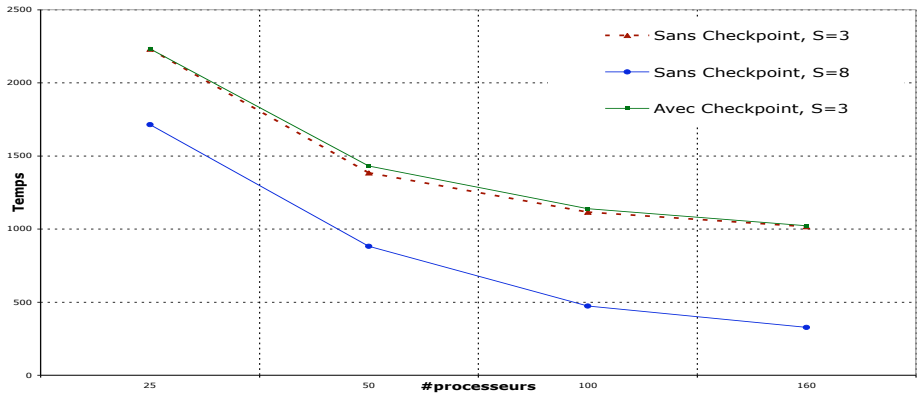



Figure 5: Nug22s sur 160 processeurs.

remarque est que le temps diminue lorsque l'on ajoute des processeurs. Néanmoins pour un seuil d'arrêt de génération du parallélisme supérieur à 3, le support stable (ici des fichiers sur NFS) devient défaillant (!) : un trop grand nombre de nœuds tentent d'écrire sur le serveur NFS qui s'effondre. Pour 160 processeurs (80 nœuds de la grappe) et pour un seuil d'arrêt de 3, le surcoût dû au protocole de sauvegarde est faible ($0.5\%$) mais la perte de parallélisme est très importante : à cause de NFS, la version avec protocole de sauvegarde est 3 fois plus lente que la version sans pour un seuil plus important.

## 7. Conclusions

Dans cet article, nous avons présenté et comparé deux protocoles de sauvegarde pour l'implantation d'un mécanisme de tolérance aux pannes dans un intergiciel pour grille. Nous avons proposé un modèle de coût qui tient compte des opérations nécessaires pour la sauvegarde et nous avons donné le temps d'exécution des programmes de type multiflots en tenant compte d'une panne durant l'exécution. Des expérimentations jusqu'à 160 processeurs valident notre modèle. Cet article ne présente que la partie «exécution» d'un système complet pour la tolérance aux pannes et plus particulièrement l'interaction avec le support stable de stockage des fichiers de sauvegarde. Il conviendrait de compléter cette étude par le coût de la partie «contrôle» de l'exécution (détecteur de défaillances et contrôleur pour la reprise).

Nous envisageons de continuer à expérimenter notre protocole périodique (non coordonné) dont le surcoût est plus faible et davantage contrôlable sur une grille de calcul pour la récupération des cycles inutilisés. Le système complet permettra de faire migrer en cours d'exécution une application déployée sur une grille et possédera un support persistant distribué par duplication des fichiers sur les noeuds de calcul [14].

## Bibliographie

1. A. Nguyen-Tuong, A. S. Grimshaw (M. Hyett). – Exploiting data-flow for fault-tolerance in a wide-area parallel system. *In : Proceedings 15 th Symposium on Reliable Distributed Systesm*, pp. 2–11. – 1996.
2. Anstreicher (K. M.), Brixius (N. W.), Goux (J. P.) et Linderoth (J.). – *Solving large quadratic assignment problems on computational grids*. – Rapport technique, Iowa City, Iowa 52242, 2000.
3. Baude (F.), Caromel (D.), Delbé (C.) et Henrio (L.). – *A Fault Tolerance protocol for ASP calculus: Design and Proof,RR-5246,INRIA-Sophia Antipolis, Equipe : OASIS*. – Rapport technique, 2004.
4. Beaumont (O.), Daoudi (E.M), Maillard (N.), Manneback (P.) et Roch (J.-L.). – 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.

5. Blumofe (R.D.), Joerg (C.F.), Kuszmaul (B.C.), Leiserson (C.E.), Randall (K.H.) et Zhou (Y.). – Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, vol. 37, n1, 1996, pp. 55–69.
6. Blumofe (R.D.) et Leiserson (C.E.). – Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, vol. 1, n27, 1997, pp. 202–229.
7. Bosilca (G.), Bouteiller (A.), Cappello (F.), Djilali (S.), Fédak (G.), Germain (C.), Hérault (T.), Lemarinier (P.), Lodygensky (O.), Magniette (F.), Néri (V.) et Selikhov (A.). – Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. *In : SuperComputing*. – Baltimore, USA, 2002.
8. Bouteiller (A.), Cappello (F.), Hérault (T.), Lemarinier (P.), Krawezik (G.) et Magniette (F.). – Mpich-v2: a fault tolerant mpi for volatile nodes based on the pessimistic sender based message logging. *In : SuperComputing*. – Phoenix, USA, 2003.
9. Chandra (T. D.) et Toueg (S.). – Unreliable failure detectors for reliable distributed systems. *J. ACM*, vol. 43, n2, 1996, pp. 225–267.
10. E. (Heymannm), Senar (M. A.), Luque (E.) et Livny (M.). – Adaptive scheduling for master-worker applications on the computational grid. *In : Proceedings of the First IEEE/ACM International Workshop on Grid Computing (GRID 2000)*. – Bangalore, India, December 2000.
11. Elnozahy (E. N. Mootaz), Alvisi (L.), Wang (Y.-M.) et B. (Johnson D.). – A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, vol. 34, n3, 2002, pp. 375–408.
12. Fatourou (P.) et Spirakis (P.G.). – Efficient scheduling of strict multithreaded computations. *Theory of Computing Systems*, vol. 33, n3, 2000, pp. 173–232.
13. Flavin (C.). – Understanding fault-tolerant distributed systems. *Commun. ACM*, vol. 34, n2, 1991, pp. 56–78.
14. G. Zheng, L. Shi (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.
15. Galilée (F.), Roch (J.-L.), Cavalheiro (G.) et Doreille (M.). – Athapascan-1: On-line building data flow graph in a parallel language. *In : PACT'98*, éd. par IEEE, pp. 88–95. – Paris, France, octobre 1998.
16. J. Leon, L. Fisher Allan (P. Steenkiste). – *Fail-safe PVM: A Portable Package for Distributed Programming with Transparent Recovery*. – Rapport technique nCMU-CS-93-124, Feb 93.
17. Jafar (S.) et Roch (J.-L.). – 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.
18. Jafar (S.), Varrette (S.) et Roch (J.-L.). – Using data-flow analysis for resilience and result checking in peer-to-peer computations. *In : IEEE DEXA'2004*. – Zaragoza, Spain, august 2004.
19. K. M. Chandy (L. Lamport). – Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, vol. 3, n1, 1985, pp. 63–75.
20. Laprie (J. C.). – *Concepts de base de la tolérance aux fautes*. – 1994.
21. Litzkow (M.), Tannenbaum (T.), Basney (J.) et Livny (M.). – *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*. – Rapport technique nCS-TR-97-1346, Univ. Wisconsin, Madison, 1997.
22. M.Wiesmann, F. Pedonne (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)*, 1999, pp. 351–360.
23. R. Strom (S. Yemini). – Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, vol. 3, n 3, 1985, pp. 204–226.
24. Revire (R.). – *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.
25. Strumpen (V.). – *Compiler Technology for Portable Checkpoints*. – Rapport technique nMA-02139, MIT Laboratory for Computer Science, Cambridge, 1998.